Standard Template Library 18

Libraries are not made; they grow.

AUGUSTINE BIRRELL

INTRODUCTION

There is a large collection of standard data structures for holding data. Since they are so standard it makes sense to have standard portable implementations for them. The Standard Template Library (STL) includes libraries for such data structures. Included in the STL are implementations of the stack, queue, and many other standard data structures. When discussed in the context of the STL, these data structures are usually called *container classes* because they are used to hold collections of data. In Chapter 8 we presented a preview of the STL by describing the `vector` template class, which is one of the container classes in the STL. In this chapter we will present an overview of some of the basic classes included in the STL. We do not have room to give a comprehensive treatment of the STL here, but we will present enough to get you started using some basic STL container classes.

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and was based on research by Stepanov, Lee, and David Musser. It is a collection of libraries written in the C++ language. Although the STL is not part of the core C++ language, it is part of the C++ standard and so any implementation of C++ that conforms to the standard would include the STL. As a practical matter, you can consider the STL to be part of the C++ language.

As its name suggest, the classes in the STL are template classes. A typical container class in the STL has a type parameter for the type of data to be stored in the container class. The STL container classes make extensive use of iterators, which are objects that facilitate cycling through the data in a container. An introduction to the concept of an iterator was given in Section 13.1, where we discussed pointers used as iterators. You will find it helpful to read that section before reading this chapter. If you have not already done so, you should also read Section 8.3, which covers the `vector` template class of the STL.

The STL also includes implementations of many important generic algorithms, such as searching and sorting algorithms. The algorithms are implemented as template functions. After discussing the container classes, we will describe some of these algorithm implementations.

The STL differs from other C++ libraries, such as `<iostream>` for example, in that the classes and algorithms are **generic,** which is another way of saying they are template classes and template functions.

PREREQUISITES

This chapter uses the material from Chapters 2 through 13, 15, and Chapter 17.

18.1 ITERATORS

> *The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.*
>
> *"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."*
>
> LEWIS CARROLL, *Alice in Wonderland*

Vectors, introduced in Chapter 8, are one of the container template classes in the STL. Iterators are a generalization of pointers. (Chapter 13 includes an introduction to pointers used as iterators.) This section shows you how to use iterators with vectors. Other container template classes, which we introduce in Section 18.2, use iterators in the same way. So, all you learn about iterators in this section will apply across a wide range of containers and does not apply solely to vectors. This reflects one of the basic tenets of the STL philosophy: The semantics, naming, and syntax for iterator usage should be (and are) uniform across different container types. We begin with a review and discussion of the `using` declarations, which we will use extensively when discussing iterators and the STL.

`using` Declarations

It may help to review the subsection entitled "Qualifying Names" in Chapter 12 before you continue with this subsection and this chapter.

Suppose `my_function` is a function defined in the namespace `my_space`. The following `using` declaration allows you to use the identifier `my_function` and have it mean the versions of `my_function` defined in the namespace `my_space`:

```
using my_space::my_function;
```

Within the scope of this `using` declaration an expression such as `my_function(1,2)` means the same thing as `my_space::my_function(1,2);` that is, within the scope of this `using` declaration the identifier `my_function` always indicates the version of `my_function` defined in `my_space`, as opposed to any definition of `my_function` defined in any other namespace.

When discussing iterators we will often apply the `::` operator to another level. You will often see expressions such as the following:

```
using std::vector<int>::iterator;
```

In this case, the identifier `iterator` names a type. So within the scope of this *`using`* directive, the following would be allowed:

```
iterator p;
```

This declares `p` to be of the type `iterator`. What is the type `iterator`? It is defined in the definition of the class `vector<int>`. Which class `vector<int>`? The one defined in the namespace `std`. (We will fully explain the type `iterator` later. At this point we are concerned only with explaining *`using`* directives.)

You may object that this is all a big to-do about nothing. There is no class `vector<int>` defined in any namespace other than the namespace `std`. That may or may not be true, but there could be a class named `vector<int>` defined in some other namespace either now or in the future. You may object further that you never heard of defining a type within a class. We have not covered such definitions, but they are possible and they are common in the STL. So, you must know how to use such types, even if you do not define such types.

In summary, consider the *`using`* directive

```
using std::vector<int>::iterator;
```

Within the scope of this *`using`* directive the identifier `iterator` means the type named `iterator` that is defined in the class `vector<int>`, which in turn is defined in the `std` namespace.

`iterator` Basics

An **iterator** is a generalization of a pointer, and in fact is typically even implemented using a pointer, but the abstraction of an iterator is designed to spare you the details of the implementation and give you a uniform interface to iterators that is the same across different container classes. Each container class has its own iterator types, just like each data type has its own pointer type. But just as all pointer types behave essentially the same for dynamic variables of their particular data type, so too does each iterator type behave the same, but each iterator is used only with its own container class.

An iterator is not a pointer, but you will not go far wrong if you think of it and use it as if it were a pointer. Like a pointer variable, an iterator variable is located at ("points to") one data entry in the container. You manipulate iterators using the following overloaded operators that apply to iterator objects:

- Prefix and postfix increment operators, `++`, for advancing the iterator to the next data item
- Prefix and postfix decrement operators, `--`, for moving the iterator to the previous data item.
- Equal and unequal operators, `==` and `!=`, to test whether two iterators point to the same data location.

- A dereferencing operator, `*`, so that if `p` is an iterator variable, then `*p` gives access to the data located at ("pointed to by") `p`. This access may be read-only, write-only, or allow both reading and changing of the data, depending on the particular container class.

Not all iterators have all of these operators. However, the `vector` template class is an example of a container whose iterators have all these operators and more.

A container class has member functions that get the iterator process started. After all, a new iterator variable is not located at ("pointing to") any data in the container. Many container classes, including the `vector` template class, have the following member functions that return iterator objects (iterator values) that point to special data elements in the data structure:

- `c.begin()` returns an iterator for the container `c` that points to the "first" data item in the container `c`.
- `c.end()` returns something that can be used to test when an iterator has passed beyond the last data item in a container `c`. The iterator `c.end()` is completely analogous to `NULL` used to test when a pointer has passed the last node in a linked list of the kind discussed in Chapter 13. The iterator `c.end()` is thus an iterator that is located at no data item, but that is a kind of end marker or sentinel.

For many container classes, these tools allow you to write *for* loops that cycle through all the elements in a container object `c`, as follows:

```
//p is an iterator variable of the type for the container object c.
for (p = c.begin(); p != c.end(); p++)
    process *p //*p is the current data item.
```

That's the big picture. Now let's look at the details in the concrete setting of the `vector` template container class.

Display 18.1 illustrates the use of iterators with the `vector` template class. Keep in mind that each container type in the STL has its own iterator types, although they are all used in the same basic ways. The iterators we want for a vector of *int*s are of type

```
std::vector<int>::iterator
```

Another container class is the `list` template class. Iterators for `list`s of *int*s are of type

```
std::list<int>::iterator
```

In the program in Display 18.1, we specialize the type name `iterator` so that it applies to iterators for vectors of *int*s. The type name `iterator` that we want in Display 18.1 is defined in the template class `vector` and so if we specialize the template class `vector` to *int*s and want the iterator type for `vector<int>`, we want the type

```
std::vector<int>::iterator;
```

DISPLAY 18.1 Iterators Used with a Vector

```
1   //Program to demonstrate STL iterators.
2   #include <iostream>
3   #include <vector>
4   using std::cout;
5   using std::endl;
6   using std::vector;
7   int main()
8   {
9       vector<int> container;
10      for (int i = 1; i <= 4; i++)
11          container.push_back(i);
12      cout << "Here is what is in the container:\n";
13      vector<int>::iterator p;
14      for (p = container.begin(); p != container.end(); p++)
15          cout << *p << " ";
16      cout <<endl;
17      cout << "Setting entries to 0:\n";
18      for (p = container.begin(); p != container.end(); p++)
19          *p = 0;
20
21      cout << "Container now contains:\n";
22      for (p = container.begin(); p != container.end(); p++)
23          cout << *p << " ";
24      cout << endl;
25      return 0;
26  }
```

Sample Dialogue

```
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
```

Since the `vector` definition places the name `vector` in the `std` namespace, the entire *`using`* declaration is

```
using std::vector<int>::iterator;
```

The basic use of iterators with the `vector` (or any container class) is illustrated by the following lines from Display 18.1:

```
vector<int>::iterator p;
for (p = container.begin(); p != container.end(); p++)
    cout << *p << " ";
```

Recall that container is of type vector<*int*>.

A vector v can be thought of as a linear arrangement of its data elements. There is a first data element v[0], a second data element v[1], and so forth. An **iterator** p is an object that can be **located at** one of these elements. (Think of p as pointing to one of these elements.) An iterator can move its location from one element to another element. If p is located at, say, v[7], then p++ moves p so it is located at v[8]. This allows an iterator to move through the vector from the first element to the last element, but it needs to find the first element and needs to know when it has seen the last element.

You can tell if an iterator is at the same location as another iterator using the operator ==. Thus, if you have an iterator pointing to the first, last, or other element, you could test another iterator to see if it is located at the first, last, or other element.

If p1 and p2 are two iterators, then the comparison

```
p1 == p2
```

is true when and only when p1 and p2 are located at the same element. (This is analogous to pointers. If p1 and p2 were pointers, this would be true if they pointed to the same thing.) As usual, != is just the negation of == and so

```
p1 != p2
```

is true when p1 and p2 are not located at the same element.

The member function begin() is used to position an iterator at the first element in a container. For vectors, and many other container classes, the member function begin() returns an iterator located at the first element. (For a vector v the first element is v[0].) Thus,

```
vector<int>::iterator p = v.begin();
```

initializes the iterator variable p to an iterator located at the first element. So, the basic *for* loop for visiting all elements of the vector v is

```
vector<int>::iterator p;
for (p = v.begin(); Boolean_Expression>; p++)
   Action_At_Location p;
```

The desired *Boolean_Expression* for a stopping condition is

```
p == v.end()
```

The member function end() returns a sentinel value that can be checked to see if an iterator has passed the last element. If p is located at the last element, then after p++, the test p = v.end() changes from *false* to *true*. So the *for* loop with the correct *Boolean_Expression* is

```
vector<int>::iterator p;
for (p = v.begin(); p != v.end(); p++)
   Action_At_Location p;
```

Note that p != v.end() does not change from *true* to *false* until after p's location has advanced past the last element. So, v.end() is not located at any

element. The value `v.end()` is a special value that serves as a sentinel value. It is not an ordinary iterator, but you can compare `v.end()` to an iterator using `==` and `!=`. The value `v.end()` is analogous to the value `NULL` used to mark the end of a linked list of the kind discussed in Chapter 13.

The following *for* loop from Display 18.1 uses this exact technique with the vector named `container`:

```
vector<int>::iterator p;
for (p = container.begin(); p != container.end(); p++)
   cout << *p << " ";
```

The action taken at the location of the iterator `p` is

```
cout << *p << " ";
```

The dereferencing operator `*` is overloaded for STL container iterators so that `*p` produces the element at location `p`. In particular, for a vector container, `*p` produces the element located at the iterator `p`. So, the `cout` statement above outputs the element located at the iterator `p` and the entire *for* loop outputs all the elements in the vector container.

The **dereferencing operator** `*p` always produces the element located at the iterator `p`. In some situations, `*p` produces read-only access, which does not allow you to change the element. In other situations, it gives you access to the element and will let you change the element. For vectors, `*p` will allow you to change the element located at `p`, as illustrated by the following *for* loop from Display 18.1:

```
for (p = container.begin(); p != container.end(); p++)
    *p = 0;
```

This *for* loop cycles through all the elements in the vector container and changes all the elements to `0`.

PROGRAMMING TIP Use `auto` to Simplify Variable Declarations

The `auto` keyword can make your code much more readable when it comes to templates and iterators. Declaring an iterator can be really verbose:

```
vector<int>::iterator p = v.begin();
```

We can do the same thing much more compactly with `auto`:

```
auto p = v.begin();
```

PITFALL Compiler Problems

Some compilers have problems with iterator declarations. You can declare an iterator in different ways. For example, we have been using the following:

```
using std::vector;
 . . .
vector<char>::iterator p;
```

Iterator

An iterator is an object that can be used with a container to gain access to elements in the container. An iterator is a generalization of the notion of a pointer, and the operators ==, !=, ++, and -- behave the same for iterators as they do for pointers. The basic outline of how an iterator can cycle through all the elements in a container is

```
STL_Container<type>::iterator p;
for (p = container.begin(); p != container.end(); p++)
   Process_Element_At_Location p;
```

`STL_Container` is the name of the container class (for example, `vector`) and `type` is the data type of the item to be stored. The member function `begin()` returns an iterator located at the first element. The member function `end()` returns a value that serves as a sentinel value one location past the last element in the container.

Alternatively, if your code only uses a single type of iterator, you could use the following:

```
using std::vector<char>::iterator;
. . .
iterator p;
```

You also could use the following, which is not quite as nice, because it introduces all names from the `std` namespace to the current declarative region, increasing the likelihood of a name conflict.

```
using namespace std;
 . . .
vector<char>::iterator p;
```

There are other, similar variations. Your compiler should accept any of these alternatives. However, we have found that some compilers will accept only certain of them. If one form does not work with your compiler, try another. ■

Dereferencing

The dereferencing operator `*p` when applied to an iterator `p` produces the element located at the iterator `p`. For some STL container classes, `*p` produces read-only access, which does not allow you to change the element. For other STL container classes, it gives you access to the element and will let you change the element.

SELF-TEST EXERCISES

1. If v is a vector, what does `v.begin()` return? What does `v.end()` return?
2. If p is an iterator for a vector object v, what is `*p`?
3. Suppose v is a vector of *int*s. Write a *for* loop that outputs all the elements of v, except for the first element.

Kinds of Iterators

Different containers have different kinds of iterators. Iterators are classified according to the kinds of operations that work on them. Vector iterators are of the most general form; that is, all the operations work with vector iterators. So, we will again use the vector container to illustrate iterators. In this case we use a vector to illustrate the iterator operators of *decrement* and *random access*. Display 18.2 shows another program using a vector object named `container` and an iterator `p`.

DISPLAY 18.2 Bidirectional and Random Access Iterator Use *(part 1 of 2)*

```
1   //Program to demonstrate bidirectional and random access iterators.
2   #include <iostream>
3   #include <vector>
4   using std::cout;
5   using std::endl;
6   using std::vector;
7
8   int main()
9   {
10      vector<char> container;
11      container.push_back('A');
12      container.push_back('B');
13      container.push_back('C');
14      container.push_back('D');

15      for (int i = 0; i < 4; i++)
16          cout << "container[" << i << "] == "
17               << container[i] << endl;
18      vector<char>::iterator p = container.begin();
19      cout << "The third entry is " << container[2] << endl;
20      cout << "The third entry is " << p[2] << endl;
21      cout << "The third entry is " << *(p + 2) << endl;

22      cout << "Back to container[0].\n";
23      p = container.begin();
24      cout << "which has value " << *p << endl;
```

Three different notations for the same thing.

This notation is specialized to vectors and arrays.

These two work for any random access iterator.

(continued)

DISPLAY 18.2 Bidirectional and Random Access Iterator Use *(part 2 of 2)*

```
25        cout << "Two steps forward and one step back:\n";
26        p++;
27        cout << *p << endl;
28        p++;
29        cout << *p << endl;
30        p--;
31        cout << *p << endl;
32        return 0;
33    }
```

This is the decrement operator. It works for any bidirectional iterator.

Sample Dialogue

```
container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

The **decrement operator** is used in Display 18.2, where the line containing it is shown in highlight. As you would expect, `p--` moves the iterator `p` to the previous location. The decrement operator `--` is the same as the increment operator `++`, but it moves the iterator in the opposite direction.

The increment and decrement operators can be used in either prefix (`++p`) or postfix (`p++`) notation. In addition to changing `p`, they also return a value. The details of the value returned are completely analogous to what happens with the increment and decrement operators on *`int`* variables. In prefix notation, first the variable is changed and the changed value is returned. In postfix notation, the value is returned before the variable is changed. We prefer not to use the increment and decrement operators as expressions that return a value and use them only to change the variable value.

The following lines from Display 18.2 illustrate that with vector iterators you have *random access* to the elements of a vector, such as `container`:

```
vector<char>::iterator p = container.begin();
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

Random access means you can go in one step directly to any particular element. We have already used `container[2]` as a form of random access to a vector. It is simply the square bracket operator that is standard with arrays and vectors. What is new is that you can use this same square bracket notation with an iterator. The expression `p[2]` is a way to obtain access to the element indexed by `2`.

The expressions `p[2]` and `*(p + 2)` are completely equivalent. By analogy to pointer arithmetic (see Chapter 9), `(p + 2)` names the location two places beyond `p`. Since `p` is at the first (index `0`) location in the above code, `(p + 2)` is at the third (index `2`) location. The expression `(p + 2)` returns an iterator. The expression `*(p + 2)` dereferences that iterator. Of course, you can replace `2` with a different nonnegative integer to obtain a pointer pointing to a different element.

Be sure to note that neither `p[2]` nor `(p + 2)` changes the value of the iterator in the iterator variable `p`. The expression `(p + 2)` returns another iterator at another location, but it leaves `p` where it was. The same thing happens with `p[2]`. Also note that the meaning of `p[2]` and `(p + 2)` depends on the location of the iterator in `p`. For example, `(p + 2)` means two locations beyond the location of `p`, wherever that may be.

For example, suppose the previously discussed code from Display 18.2 were replaced with the following (note the added `p++`):

```
vector<char>::iterator p = container.begin();
p++;
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

The output of these three `cout`s would no longer be

```
The third entry is C
The third entry is C
The third entry is C
```

but would instead be

```
The third entry is C
The third entry is D
The third entry is D
```

The `p++` moves `p` from location `0` to location `1` and so `(p + 2)` is now an iterator at location `3`, not location `2`. So, `*(p + 2)` and `p[2]` are equivalent to `container[3]`, not `container[2]`.

Kinds of Iterators

Different containers have different kinds of iterators. The following are the main kinds of iterators:

`Forward iterators:` ++ works on the iterator.

`Bidirectional iterators:` both ++ and -- work on the iterator.

`Random access iterators:` ++, --, and random access all work with the iterator.

We now know enough about iterators to make sense of how iterators are classified. The main kinds of iterators are

Forward iterators: ++ works on the iterator.

Bidirectional iterators: both ++ and -- work on the iterator.

Random access iterators: ++, --, and random access all work with the iterator.

Note that these are increasingly strong categories: Every random access iterator is also a bidirectional iterator, and every bidirectional iterator is also a forward iterator. As we will see, different template container classes have different kinds of iterators. The iterators for the `vector` template class are random access iterators.

Note that the names *forward iterator*, *bidirectional iterator*, and *random access iterator* refer to kinds of iterators, not type names. The actual type names will be something like `std::vector<int>::iterator`, which in this case happens to be a random access iterator.

SELF-TEST EXERCISE

4. Suppose the vector `v` contains the letters `'A'`, `'B'`, `'C'`, and `'D'` in that order. What is the output of the following code?

```
vector<char>::iterator i = v.begin();
i++;
cout << *(i + 2) << " ";
i--;
cout << i[2] << " ";
cout << *(i + 2) << " ";
```

Constant and Mutable Iterators

The categories forward iterator, bidirectional iterator, and random access iterator each subdivide into two categories: *constant* and *mutable*, depending on how the dereferencing operator behaves with the iterator. With a **constant iterator** the dereferencing operator produces a read-only version of the element. With a constant iterator `p`, you can use `*p`, for example, to assign it to a variable or output it to the screen, but you cannot change the element in the container by, for example, assigning it to `*p`. With a **mutable iterator p,** `*p` can be assigned a value and that will change the corresponding element in the container. The vector iterators are mutable, as shown by the following lines from Display 18.1:

```
cout << "Setting entries to 0:\n";
for (p = container.begin(); p != container.end(); p++)
    *p = 0;
```

If a container has only constant iterators, you cannot obtain a mutable iterator for the container. However, if a container has mutable iterators and you want a constant iterator for the container, you can have it. You might want a constant iterator as a kind of error checking if you intend that your code not change the elements in the container. For example, the following will produce a constant iterator for a vector container named `container`:

```
std::vector<char>::const_iterator p = container.begin();
```

or equivalently

```
using std::vector<char>::const_iterator;
const_iterator p = container.begin();
```

With `p` declared in this way, the following would produce an error message:

```
*p = 'Z';
```

For example, Display 18.2 would behave exactly the same if you change

```
vector<int>::iterator p;
```

to

```
vector<int>::const_iterator p;
```

However, a similar change would not work in Display 18.1 because of the following line from the program in Display 18.1:

```
*p = 0;
```

Note that `const_iterator` is a type name, while *constant iterator* is the name of a kind of iterator. However, every iterator of a type named `const_iterator` will be a constant iterator.

Constant Iterator

A constant iterator is an iterator that does not allow you to change the element at its location.

Reverse Iterators

Sometimes you want to cycle through the elements in a container in reverse order. If you have a container with bidirectional iterators, you might be tempted to try

```
vector<int>::iterator p;
for (p = container.end(); p != container.begin(); p--)
    cout << *p << " ";
```

This code will compile, and you may be able to get something like this to work on some systems, but there is something fundamentally wrong with this: `container.end()` is not a regular iterator, but only a sentinel, and `container.begin()` is not a sentinel.

Fortunately, there is an easy way to do what you want. For a container with bidirectional iterators, there is a way to reverse everything using a kind of iterator known as a **reverse iterator.** The following will work fine:

```
vector<int>::reverse_iterator rp;
for (rp = container.rbegin(); rp != container.rend(); rp++)
    cout << *rp << " ";
```

The member function `rbegin()` returns an iterator located at the last element. The member function `rend()` returns a sentinel that marks the "end" of the elements in the reverse order. Note that for an iterator of type `reverse_iterator`, the increment operator `++` moves backward through the elements. In other words, the meanings of `--` and `++` are interchanged. The program in Display 18.3 demonstrates a reverse iterator.

Reverse Iterators

A reverse iterator can be used to cycle through all elements of a container, provided that the container has bidirectional iterators. The general scheme is as follows:

```
STL_Container<type>::reverse_iterator rp;
for (rp = c.rbegin(); rp != c.rend(); rp++)
    Process_At_Location rp;
```

The object `c` is a container class with bidirectional iterators.

DISPLAY 18.3 Reverse Iterator

```
1   //Program to demonstrate a reverse iterator.
2   #include <iostream>
3   #include <vector>
4   using std::cout;
5   using std::endl;
6   using std::vector;
7   int main()
8   {
9       vector<char> container;
10      container.push_back('A');
11      container.push_back('B');
12      container.push_back('C');
13      cout << "Forward:\n";
14      vector<char>::iterator p;
15      for (p = container.begin(); p != container.end(); p++)
16          cout << *p << " ";
17      cout << endl;
18
19      cout << "Reverse:\n";
20      vector<char>::reverse_iterator rp;
21      for (rp = container.rbegin(); rp != container.rend(); rp++)
22          cout << *rp << " ";
23      cout << endl;
24      return 0;
25  }
```

Sample Dialogue

```
Forward:
A B C
Reverse:
C B A
```

The `reverse_iterator` type also has a constant version, which is named `const_reverse_iterator`.

Other Kinds of Iterators

There are other kinds of iterators that we will not cover in this book. Briefly, two kinds of iterators you may encounter are an **input iterator,** which is essentially a forward iterator that can be used with input streams, and an

output iterator, which is essentially a forward iterator that can be used with output streams. For more details, you will need to consult a more advanced reference.

SELF-TEST EXERCISES

5. Suppose the vector v contains the letters 'A', 'B', 'C', and 'D' in that order. What is the output of the following code?

```
vector<char>::reverse_iterator i = v.rbegin();
i++;
i++;
cout << *i << " ";
i--;
cout << *i << " ";
```

6. Suppose you want to run the following code, where v is a vector of *int*s:

```
for (p = v.begin(); p != v.end(); p++)
    cout << *p << " ";
```

Which of the following are possible ways to declare p?

```
std::vector<int>::iterator p;
std::vector<int>::const_iterator p;
```

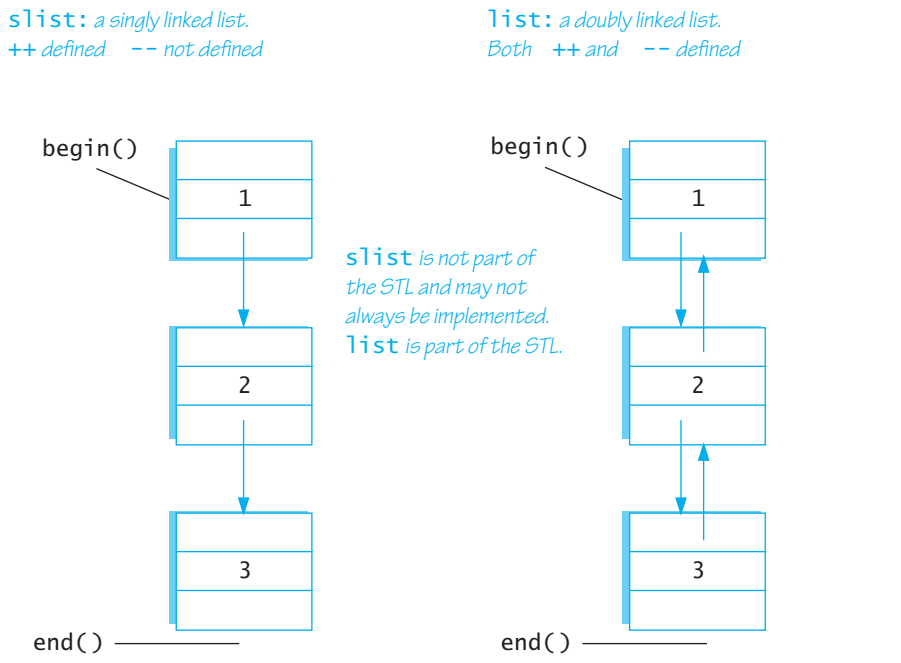
18.2 CONTAINERS

Put all your eggs in one basket and
—WATCH THAT BASKET.

MARK TWAIN, *Pudd'n head Wilson*

The **container classes** of the STL are different kinds of data structures for holding data, such as lists, queues, and stacks. Each is a template class with a parameter for the particular type of data to be stored. So, for example, you can specify a list to be a list of *int*s, or *double*s, or `string`s, or any class or `struct` type you wish. Each container template class may have its own specialized accessor and mutator functions for adding data and removing data from the container. Different container classes may have different kinds of iterators. For example, one container class may have bidirectional iterators while another container class may have only forward iterators. However, whenever they are defined the iterator operators and the member functions `begin()` and `end()` have the same meaning for all STL container classes.

DISPLAY 18.4 Two Kinds of Lists



Sequential Containers

A sequential container arranges its data items into a list so that there is a first element, a next element, and so forth up to a last element. The linked lists we discussed in Chapter 13 are examples of a kind of list. The lists we discussed in Chapter 13 are sometimes called **singly linked lists** because there is only one link from one location to another. The STL has no container corresponding to such singly linked lists, although some implementations do offer an implementation of them, typically under the name `slist`. The simplest list that is part of the STL is the **doubly linked list,** which is the template class named `list`. The difference between these two kinds of lists is illustrated in Display 18.4.

The lists in Display 18.4 contain the three integer values `1`, `2`, and `3` in that order. The types for the two lists are `slist<int>` and `list<int>`. That display also indicates the location of the iterators `begin()` and `end()`. We have not yet told you how you can enter the integers into the lists.

In Display 18.4 we have drawn our singly and doubly linked lists as nodes and pointers of the form discussed in Chapter 14. The STL class `list` and the nonstandard class `slist` might (or might not) be implemented in this way.

However, when using the STL template classes, you are shielded from these implementation details. So, you simply think in terms of locations for the data (which may or may not be nodes) and iterators (not pointers). You can think of the arrows in Display 18.4 as indicating the directions for ++ (which is down) and -- (which is up in Display 18.4).

We wanted to present the template class `slist` to help give a context for the sequential containers. It corresponds to what we discussed most in Chapter 13, and it is the first thing that comes to the mind of most programmers when you mention *linked lists*. However, since the template class `slist` is not standard, we will discuss it no more. If your implementation offers the template class `slist` and you want to use it, the details are similar to those we will describe for `list`, except that the decrement operators -- (prefix and postfix) are not defined for `slist`.

A simple program using the STL template class `list` is given in Display 18.5. The function `push_back` adds an element to the end of the list. Notice that for the `list` template class, the dereferencing operator gives you access to the data for reading and for changing the data. Also notice that with the `list` template class and all the template classes and iterators of the STL, all definitions are placed in the `std` namespace.

DISPLAY 18.5 Using the `list` Template Class *(part 1 of 2)*

```
1   //Program to demonstrate the STL template class list.
2   #include <iostream>
3   #include <list>
4   using std::cout;
5   using std::endl;
6   using std::list;
7
8   int main()
9   {
10      list<int> list_object;
11
12      for (int i = 1; i <= 3; i++)
13          list_object.push_back(i);
14
15      cout << "List contains:\n";
16      list<int>::iterator iter;
17      for (iter = list_object.begin(); iter != list_object.end(); iter++)
18          cout << *iter << " ";
19      cout << endl;
20
21      cout << "Setting all entries to 0:\n";
```

(continued)

DISPLAY 18.5 Using the `list` Template Class *(part 2 of 2)*

```
22        for (iter = list_object.begin(); iter != list_object.end(); iter++)
23            *iter = 0;
24
25        cout << "List now contains:\n";
26        for (iter = list_object.begin(); iter != list_object.end(); iter++)
27            cout << *iter << " ";
28        cout << endl;
29
30        return 0;
31    }
```

Sample Dialogue

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

Note that Display 18.5 would compile and run exactly the same if we replace `list` and `list<int>` with `vector` and `vector<int>`, respectively. This uniformity of usage is a key part of the STL syntax.

There are, however, differences between a vector and a list container. One of the main differences is that a vector container has random access iterators while a list has only bidirectional iterators. For example, if you start with Display 18.2, which uses random access, and replace all occurrences of `vector` and `vector<char>` with `list` and `list<char>`, respectively, and then compile the program, you will get a compiler error. (You will get an error message even if you delete the statements containing `container[i]` or `container[2]`.)

The basic sequential container template classes of the STL are given in Display 18.6. A sample of some member functions is given in Display 18.7. Other containers, such as stacks and queues, can be obtained from these using techniques discussed in the subsection entitled "Container Adapters `stack` and `queue`." All these sequence template classes have a destructor that returns storage for recycling.

Deque, pronounced "d-queue" or "deck," stands for "doubly ended queue." A deque is a kind of super queue. With a queue you add data at one end of the data sequence and remove data from the other end. With a deque

DISPLAY 18.6 STL Basic Sequential Containers

Template Class Name	Iterator Type Names	Kind of Iterators	Library Header File
slist	slist<T>::iterator	mutable forward	<slist>
Warning: slist is not part of the STL.	slist<T>::const_iterator	constant forward	Depends on implementation and may not be available.
list	list<T>::iterator list<T>::const_iterator list<T>::reverse_iterator list<T>::const_reverse_iterator	mutable bidirectional constant bidirectional mutable bidirectional constant bidirectional	<list>
vector	vector<T>::iterator vector<T>::const_iterator vector<T>::reverse_iterator vector<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<vector>
deque	deque<T>::iterator deque<T>::const_iterator deque<T>::reverse_iterator deque<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<deque>

DISPLAY 18.7 Some Sequential Container Member Functions *(part 1 of 2)*

Member Function (c is a Container Object)	Meaning
c.size()	Returns the number of elements in the container.
c.begin()	Returns an iterator located at the first element in the container.
c.end()	Returns an iterator located one beyond the last element in the container.
c.rbegin()	Returns an iterator located at the last element in the container. Used with reverse_iterator. Not a member of slist.
c.rend()	Returns an iterator located one beyond the first element in the container. Used with reverse_iterator. Not a member of slist.
c.push_back(*Element*)	Insert the *Element* at the end of the sequence. Not a member of slist.

(continued)

DISPLAY 18.7 Some Sequential Container Member Functions *(part 2 of 2)*

c.push_front(*Element*)	Insert the *Element* at the front of the sequence. Not a member of vector.
c.insert(Iterator, *Element*)	Insert a copy of *Element* before the location of *Iterator*.
c.erase(*Iterator*)	Removes the element at location Iterator. Returns an iterator at the location immediately following. Returns c.end() if the last element is removed.
c.clear()	A void function that removes all the elements in the container.
c.front()	Returns a reference to the element in the front of the sequence. Equivalent to *(c.begin()).
c1 == c2	True if c1.size() == c2.size() and each element of c1 is equal to the corresponding element of c2.
c1 != c2	!(c1 == c2)
<All the sequential containers discussed in this section also have a default constructor, a copy constructor, and various other constructors for initializing the container to default or specified elements. Each also has a destructor that returns all storage for recycling and a well-behaved assignment operator.>	

you can add data at either end and remove data from either end. The template class deque is a template class for a deque with a parameter for the type of data stored.

Sequential Containers

A sequential container arranges its data items into a list so that there is a first element, a next element, and so forth up to a last element. The sequential container template classes that we have discussed are `slist`, `list`, `vector`, and deque.

PITFALL Iterators and Removing Elements

When you add or remove an element to or from a container, that can affect other iterators. In general, there is no guarantee that the iterators will be located at the same element after an addition or deletion. Some containers do, however, guarantee that the iterators will not be moved by additions or deletions, except of course if the iterator is located at an element that is removed.

Of the template classes we have seen so far, `list` and `slist` guarantee that their iterators will not be moved by additions or deletions, except of course if the iterator is located at an element that is removed. The template classes `vector` and `deque` make no such guarantee.

PROGRAMMING TIP Type Definitions in Containers

The STL container classes contain type definitions that can be handy when programming with these classes. We have already seen that STL container classes may contain the type names `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator` (and hence must contain their type definitions behind the scenes). There are typically other type definitions as well.

All the template classes we have discussed so far have the defined types `value_type` and `size_type`. The type `value_type` is the type of the elements stored in the container. For example, `list<int>::value_type` is another name for *`int`*. Another defined type is `size_type`, which is an unsigned integer type that is the return type for the member function `size`. As we noted in Chapter 8, the `size_type` for the vector template class is *`unsigned int`*, although most compilers will be happy if you think of the type as just plain *`int`*.

SELF-TEST EXERCISES

7. What is a major difference between a `vector` and a `list`?

8. Which of the template classes `slist`, `list`, `vector`, and `deque` have the member function `push_back`?

9. Which of the template classes `slist`, `list`, `vector`, and `deque` have random access iterators?

10. Which of the template classes `slist`, `list`, `vector`, and `deque` can have mutable iterators?

Container Adapters `stack` and `queue`

Container adapters are template classes that are implemented on top of other classes. For example, the `stack` template class is by default implemented on top of the `deque` template class, which means that buried in the implementation of the `stack` is a `deque`, which is where all the data resides. However, you are shielded from this implementation detail and see a stack as a simple last-in/first-out data structure.

Other container adapter classes are the `queue` and `priority_queue` template classes. Stacks and queues were discussed in Chapter 13. A **priority queue** is like a queue with the additional property that each entry is given a priority when it is added to the queue. If all entries have the same priority, then entries are removed from a priority queue in the same manner as they are removed from a queue. If items have different priorities, the higher-priority items are removed before lower-priority items. We will not be discussing priority queues in any detail, but mention it for those who may be familiar with the concept.

Although an adapter template class has a default container class on top of which it is built, you may choose to specify a different underlying container, for efficiency or other reasons depending on your application. For example, any sequential container may serve as the underlying container for a `stack` and any sequential container other than `vector` may serve as the underlying container for a `queue`. The default underlying data structure is the `deque` for both the `stack` and the `queue`. For a `priority_queue`, the default underlying container is a `vector`. If you are happy with the default underlying container type, then a container adapter looks like any other template container class to you. For example, the type name for the `stack` template class using the default underlying container is `stack<int>` for a stack of `int`s. If you wish to specify that the underlying container is instead the `vector` template class, you would use `stack<int, vector<int>>` as the type name. We will always use the default underlying container.

Warning

If you do specify an underlying container, be warned that C++ compilers prior to C++11 cannot compile code with two > symbols in the type expression without a space in between them. Use `stack<int, vector<int> >`, with a space between the last two >'s. Do not use `stack<int, vector<int>>`. C++11 compilers do not need a space between the two > symbols.

The member functions and other details about the `stack` template class are given in Display 18.8. For the `queue` template class these details are given in Display 18.9. A simple example of using the `stack` template class is given in Display 18.10.

DISPLAY 18.8 Stack Template Class *(part 1 of 2)*

`Stack` Adapter Template Class Details

Type name `stack<T>` or `stack<T, Underlying_Container>` for a stack of elements of type `T`.
Library header: `<stack>`, which places the definition in the `std` namespace.
Defined types: `value_type, size_type.`
There are no iterators.

(continued)

DISPLAY 18.8 Stack Template Class *(part 2 of 2)*

Sample Member Functions

Member Function (s is a Stack Object)	Meaning
s.size()	Returns the number of elements in the stack.
s.empty()	Returns *true* if the stack is empty; otherwise returns *false*.
s.top()	Returns a mutable reference to the top member of the stack.
s.push(*Element*)	Inserts a copy of *Element* at the top of the stack.
s.pop()	Removes the top element of the stack. Note that pop is a void function. It does not return the element removed.
s1 == s2	True if s1.size() == s2.size() and each element of s1 is equal to the corresponding element of s2; otherwise returns *false*.

The stack template class also has a default constructor, a copy constructor, as well as a constructor that takes an object of any sequential container class and initializes the stack to the elements in the sequence. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

DISPLAY 18.9 Queue Template Class *(part 1 of 2)*

`Queue` **Adapter Template Class Details**
Type name `queue<T>` or `queue<T, Underlying_Container>` for a queue of elements of type `T`.
For efficiency reasons, the Underlying_Container cannot be a vector type.
Library header: `<queue>` which places the definition in the `std` namespace.
Defined types: `value_type, size_type.`
There are no iterators.

Sample Member Functions

Member Function (q is a Queue Object)	Meaning
q.size()	Returns the number of elements in the queue.
q.empty()	Returns *true* if the queue is empty; otherwise returns *false*.

(continued)

DISPLAY 18.9 Queue Template Class *(part 2 of 2)*

q.front()	Returns a mutable reference to the front member of the queue.
q.back()	Returns a mutable reference to the last member of the queue.
q.push(*Element*)	Adds *Element* to the back of the queue.
q.pop()	Removes the front element of the queue. Note that pop is a void function. It does not return the element removed.
q1 == q2	True if q1.size() == q2.size() and each element of q1 is equal to the corresponding element of q2; otherwise returns *false*.

The queue template class also has a default constructor, a copy constructor, as well as a constructor that takes an object of any sequential container class and initializes the stack to the elements in the sequence. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

DISPLAY 18.10 Program Using the Stack Template Class *(part 1 of 2)*

```
1   //Program to demonstrate the use of the stack template class from the STL.
2   #include <iostream>
3   #include stack>
4   using std::cin;
5   using std::cout;
6   using std::endl;
7   using std::stack;
8
9   int main()
10  {
11      stack<char> s;
12
13      cout << "Enter a line of text:\n";
14      char next;
15      cin.get(next);
16      while (next != '\n')
17      {
18          s.push(next);
19          cin.get(next);
20      }
21
22      cout << "Written backward that is:\n";
23      while ( !s.empty() )
24      {
```

(continued)

DISPLAY 18.10 Program Using the Stack Template Class *(part 2 of 2)*

```
25              cout << s.top();
26              s.pop();
27          }
28          cout << endl;
29
30          return 0;
31      }
```

The member function `pop` *removes one element, but does not return that element.* `pop` *is a* `void` *function. So, we needed to use* `top` *to read the element we remove.*

Sample Dialogue

```
Enter a line of text:
straw
Written backward that is:
warts
```

SELF-TEST EXERCISES

11. What kind of iterators (forward, bidirectional, or random access) does the `stack` template adapter class have?

12. What kind of iterators (forward, bidirectional, or random access) does the `queue` template adapter class have?

13. If `s` is a `stack<char>`, what is the type of the returned value of `s.pop()`?

Associative Containers `set` and `map`

Associative containers are basically very simple databases. They store data, such as `structs` or any other type of data. Each data item has an associated value known as its **key.** For example, if the data is a `struct` with an employee's record, the key might be the employee's Social Security number. Items are retrieved on the basis of the key. The key type and the type for data to be stored need not have any relationship to one another, although they often are related. A very simple case is when the each data item is its own key. For example, in a `set` every element is its own key.

The `set` template class is, in some sense, the simplest container you can imagine. It stores elements without repetition. The first insertion places an element in the set. Additional insertions after the first have no effect, so no element appears more than once. Each element is its own key; basically, you

just add or delete elements and ask if an element is in the `set` or not. Like all STL classes, the `set` template class was written with efficiency as a goal. In order to work efficiently, a `set` object stores its values in sorted order. You can specify the order used for storing elements as follows:

```
set<T, Ordering> s;
```

Ordering should be a well-behaved ordering relation that takes two arguments of type `T` and returns a *`bool`* value.[1] `T` is the type of elements stored. If no ordering is specified, then the ordering is assumed to be the < relational operator. Some basic details about the `set` template class are given in Display 18.11. A simple example that shows how to use some of the member functions of the template class `set` is given in Display 18.12.

A **map** is essentially a function given as a set of ordered pairs. For each value `first` that appears in a pair, there is at most one value `second` such that the pair `(first, second)` is in the map. The template class `map` implements map objects in the STL. For example, if you want to assign a unique number to each string name, you could declare a `map` object as follows:

```
map<string, int> number_map;
```

For `string` values known as *keys*, the `number_map` object can associate a unique *`int`* value.

An alternate way to think of a map is as an **associative array.** A traditional array maps from a numerical index to a value. For example, `a[10] = 5` would store the number 5 at index 10. An associative array allows you to define your own indices using the data type of your choice. For example, `numberMap["c++"] = 5` would associate the integer 5 with the string "`c++`". For convenience, the `[]` square bracket operator is defined to allow you to use an array-like notation to access a map, although you also can use the `insert` or `find` methods if you want.

Like a `set` object, a `map` object stores its elements in sorted order by its key values. You can specify the ordering on keys as a third entry in the angular brackets <>. If you do not specify an ordering, a default ordering is used. The restrictions on orderings you can use is the same as those on the orderings allowed for the `set` template class. Note that the ordering is on key values only. The second type can be any type and need not have anything to do with any ordering. As with the `set` object, the sorting of the stored entries in a `map` object is done for reasons of efficiency.

---

[1] The ordering must be a *strict weak ordering*. Most typical orderings used to implement the < operator is strict weak ordering. For those who want the details: A **strict weak ordering** must be: (irreflexive) `Ordering(x, x)` is always false; (antisymmetric) `Ordering(x, y)` implies `!Ordering(y, x)`; (transitive) `Ordering(x, y)` and `Ordering(y, z)` imply `Ordering(x, z)`; and (transitivity of equivalence) if `x` is equivalent to `y` and `y` is equivalent to `z`, then `x` is equivalent to `z`. Two elements `x` and `y` are equivalent if `Ordering(x, y)`and `Ordering(y, x)` are both false.

DISPLAY 18.11 set Template Class

set Template Class Details

Type name `set<T>` or `set<T, Ordering>` for a set of elements of type `T`. The `Ordering` is used to sort elements for storage. If no `Ordering` is given, the ordering used is the binary operator <.

Library header: `<set>`, which places the definition in the `std` namespace.

Defined types include: `value_type`, `size_type`.

Iterators: `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`. All iterators are bidirectional and those not including `const_` are mutable. `begin()`, `end()`, `rbegin()`, and `rend()` have the expected behavior. Adding or deleting elements does not affect iterators, except for an iterator located at the element removed.

Sample Member Functions

Member Function (s is a Set Object)	Meaning
s.insert(*Element*)	Inserts a copy of *Element* in the set. If *Element* is already in the set, this has no effect.
s.erase(*Element*)	Removes *Element* from the set. If *Element* is not in the set, this has no effect.
s.find(*Element*)	Returns a mutable iterator located at the copy of *Element* in the set. If *Element* is not in the set, s.end() is returned.
s.erase(*Iterator*)	Erases the element at the location of the *Iterator*.
s.size()	Returns the number of elements in the set.
s.empty()	Returns *true* if the set is empty; otherwise returns *false*.
s1 == s2	Returns *true* if the sets contains the same elements; otherwise returns *false*.

The set template class also has a default constructor, a copy constructor, as well as other specialized constructors not mentioned here. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

DISPLAY 18.12 Program Using the set Template Class

```
1   //Program to demonstrate use of the set template class.
2   #include <iostream>
3   #include <set>
4   using std::cout;
5   using std::endl;
6   using std::set;
7   int main()
8   {
9       set<char> s;
10
11      s.insert('A');
12      s.insert('D');
13      s.insert('D');
14      s.insert('C');
15      s.insert('C');
16      s.insert('B');
17
18      cout << "The set contains:\n";
19      set<char>::const_iterator p;
20      for (p = s.begin(); p != s.end(); p++)
21          cout << *p << " ";
22      cout << endl;
23
24      cout << "Removing C.\n";
25      s.erase('C');
26      for (p = s.begin(); p != s.end(); p++)
27          cout << *p << " ";
28      cout << endl;
29
30      return 0;
31  }
```

No matter how many times you add an element to a set, the set contains only one copy of that element.

Sample Dialogue

```
The set contains:
A B C D
Removing C.
A B D
```

The easiest way to add and retrieve data from a map is to use the `[]` operator. Given a map object `m`, the expression `m[key]` will return a reference to the data element associated with `key`. If no entry exists in the map for `key`, then a new entry will be created with the default value for the data element.

For numeric data types, the default value is 0. For objects of type string, the default value is an empty string.

The [] operator can be used to add a new item to the map or to replace an existing entry. For example, the statement `m[key] = newData;` will create a new association between `key` and `newData`. Note that care must be taken to ensure that map entries are not created by mistake. For example, if you execute the statement `val = m[key];` with the intention of retrieving the value associated with `key` but mistakenly enter a value for `key` that is not already in the map, then a new entry will be made for `key` with the default value and assigned into `val`.

Some basic details about the `map` template class are given in Display 18.13. In order to understand these details, you first need to know something about the `pair` template class.

The STL template class `pair<T1,T2>` has objects that are pairs of values such that the first element is of type `T1` and the second is of type `T2`. If `aPair` is an object of type `pair<T1,T2>`, then `aPair.first` is the first element, which is of type `T1`, and `aPair.second` is the second element, which is of type `T2`. The member variables `first` and `second` are public member variables, so no accessor or mutator functions are needed.

The header file for the `pair` template is `<utility>`. So, to use the `pair` template class, you need the following, or something like it, in your file:

```
#include <utility>
using std::pair;
```

The `map` template class uses the `pair` template class to store the association between the key and a data item. For example, given the definition

```
map<string, int> numberMap;
```

we can add a mapping from "c++" to the number 10 by using a `pair` object:

```
pair<string, int> toInsert("c++", 10);
numberMap.insert(toInsert);
```

or by using the [] operator:

```
numberMap["c++"] = 10;
```

In either case, when we access this pair using an iterator, `iterator->first` will refer to the key "c++" while `iterator->second` will refer to the data value 10. A simple example that shows how to use some of the member functions of the template class map is given in Display 18.14.

We will mention two other associative containers, although we will not give any details about them. The template classes `multiset` and `multimap` are essentially the same as `set` and `map`, respectively, except that a `multiset` allows repetition of elements and a `multimap` allows multiple values to be associated with each key value.

DISPLAY 18.13 map Template Class

map Template Class Details

Type name map<KeyType, T> or map<KeyType, T, Ordering> for a map that associates ("maps") elements of type KeyType to elements of type T.
The Ordering is used to sort elements by key value for efficient storage.
If no Ordering is given, the ordering used is the binary operator <.

Library header: <map> places the definition in the std namespace.

Defined types include: key_type for the type of the key values, mapped_type for the type of the values mapped to, and size_type. (So, the defined type key_type is simply what we called KeyType earlier.)

Iterators: iterator, const_iterator, reverse_iterator, and const_reverse_iterator. All iterators are bidirectional. Those iterators not including const_ are neither constant nor mutable, but something in between. For example, if p is of type iterator, then you change the key value but not the value of type T. Perhaps it is best, at least at first, to treat all iterators as if they were constant.
begin(), end(), rbegin(), and rend() have the expected behavior. Adding or deleting elements does not affect iterators, except for an iterator located at the element removed.

Sample Member Functions

Member Function (m is a Map Object)	Meaning
m.insert(*Element*)	Inserts *Element* in the map. *Element* is of type pair<KeyType, T>. Returns a value of type pair<iterator, *bool*>. If the insertion is successful, the second part of the returned pair is true and the iterator is located at the inserted element.
m.erase(*Target_Key*)	Removes the element with the key *Target_Key*.
m.find(*Target_Key*)	Returns an iterator located at the element with key value *Target_Key*. Returns m.end() if there is no such element.
m[*Target_Key*]	Returns a reference to the object associated with the key *Target_Key*. If the map does not already contain such an object, then a default object of type T is inserted and returned.
m.size()	Returns the number of pairs in the map.
m.empty()	Returns *true* if the map is empty; otherwise returns *false*.
m1 == m2	Returns *true* if the maps contains the same pairs; otherwise returns *false*.

The map template class also has a default constructor, a copy constructor, as well as other specialized constructors not mentioned here. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

DISPLAY 18.14 Program Using the `map` Template Class *(part 1 of 2)*

```
1   //Program to demonstrate use of the map template class.
2   #include <iostream>
3   #include <map>
4   #include <string>
5   using std::cout;
6   using std::endl;
7   using std::map;
8   using std::string;

9   int main()
10  {
11      map<string, string> planets;

12      planets["Mercury"] = "Hot planet";
13      planets["Venus"] = "Atmosphere of sulfuric acid";
14      planets["Earth"] = "Home";
15      planets["Mars"] = "The Red Planet";
16      planets["Jupiter"] = "Largest planet in our solar system";
17      planets["Saturn"] = "Has rings";
18      planets["Uranus"] = "Tilts on its side";
19      planets["Neptune"] = "1500 mile-per-hour winds";
20      planets["Pluto"] = "Dwarf planet";
21      cout << "Entry for Mercury - " << planets["Mercury"]
22           << endl << endl;
23      if (planets.find("Mercury") != planets.end())
24          cout << "Mercury is in the map." << endl;
25      if (planets.find("Ceres") == planets.end())
26          cout << "Ceres is not in the map." << endl << endl;
27      cout << "Iterating through all planets: " << endl;
28      map<string, string>::const_iterator iter;
29      for (iter = planets.begin(); iter != planets.end(); iter++)
30      {
31          cout << iter->first << " - " << iter->second << endl;
32      }
33      return 0;
34  }
```

Sample Dialogue

```
Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
```

The iterator will output the map in order sorted by the key. In this case the output will be listed alphabetically by planet.

(continued)

DISPLAY 18.14 Program Using the `map` Template Class *(part 2 of 2)*

```
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile-per-hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

VideoNote
C++11 and Containers

PROGRAMMING TIP Use Initialization, Ranged `for`, and `auto` with Containers

Several features introduced in C++11 make it easier to work with collections. In particular, you can initialize your container objects using the uniform initializer list format, which consists of initial data in curly braces. You can also use `auto` and the ranged `for` loop to easily iterate through a container. Consider the following two initialized collection objects:

```
map<int, string> personIDs = {
                                  {1,"Walt"},
                                  {2,"Kenrick"}
                              };
set<string> colors = {"red","green","blue"};
```

We can iterate through each container conveniently using a ranged `for` loop and `auto`:

```
for (auto p : personIDs)
     cout << p.first << " " << p.second << endl;
for (auto p : colors)
     cout << p << " ";
```

The output of this snippet is:

```
1 Walt
2 Kenrick
blue green red
```

Efficiency

The STL was designed with efficiency as an important consideration. In fact, the STL implementations strive to be optimally efficient. For example, the `set` and `map` elements are stored in sorted order so that algorithms that search for the elements can be more efficient.

Each of the member functions for each of the template classes has a guaranteed maximum running time. These maximum running times are expressed using what is called big-*O* notation, which we discuss in Section 18.3.

(Section 18.3 also gives some guaranteed running times for some of the container member functions we have already discussed. These are in the subsection entitled "Container Access Running Times.") When using more advanced references or even later in this chapter, you will be told the guaranteed maximum running times for certain functions.

SELF-TEST EXERCISES

14. How many elements will be in the map `mymap` after the following code is executed?

```
map<int, string> mymap;
mymap[5] = "c++";
cout << mymap[4] << endl;
```

15. Can a `set` have elements of a class type?

16. Suppose `s` is of the type `set<char>`. What value is returned by `s.find('A')` if `'A'` is in `s`? What value is returned if `'A'` is not in `s`?

18.3 GENERIC ALGORITHMS

"Cures consumption, anemia, sexual dysfunction, and all other diseases."

TYPICAL CLAIM BY A TRAVELING SALESMAN OF "SNAKE OIL"

This section covers some basic function templates in the STL. We cannot give you a comprehensive description of them all here, but will present a large enough sample to give you a good feel for what is contained in the STL and to give you sufficient detail to start using these template functions.

These template functions are sometimes called **generic algorithms.** The term *algorithm* is used for a reason. Recall that an algorithm is just a set of instructions for performing a task. An algorithm can be presented in any language, including a programming language like C++. But when using the word *algorithm*, programmers typically have in mind a less formal presentation given in English or pseudocode. As such, it is often thought of as an abstraction of the code defining a function. It gives the important details but not the fine details of the coding. The STL specifies certain details about the algorithms underlying the STL template functions and that is why they are sometimes called generic *algorithms*.

These STL function templates do more than just deliver a value in any way that the implementers wish. The function templates in the STL come with minimum requirements that must be satisfied by their implementations if they are to satisfy the standard. In most cases, they must be implemented with a guaranteed running time. This adds an entirely new dimension to the idea of a function interface. In the STL, the interface not only tells a programmer what the function does and how to use the functions; the interface also tells how rapidly the task will be done. In some cases, the standard even specifies the

particular algorithm that is used, although not the exact detail of the coding. Moreover, when it does specify the particular algorithm, it does so because of the known efficiency of the algorithm. The key new point is a specification of an efficiency guarantee for the code. In this chapter we will use the terms *generic algorithm*, *generic function*, and *STL function template* to all mean the same thing.

In order to have some terminology to discuss the efficiency of these template functions or generic algorithms, we first present some background on how the efficiency of algorithms is usually measured.

Running Times and Big-O Notation

If you ask a programmer how fast his or her program is, you might expect an answer like "two seconds." However, the speed of a program cannot be given by a single number. A program will typically take a longer amount of time on larger inputs than it will on smaller inputs. You would expect that a program to sort numbers would take less time to sort ten numbers than it would to sort one thousand numbers. Perhaps it takes two seconds to sort ten numbers, but ten seconds to sort one thousand numbers. How, then, should the programmer answer the question, "How fast is your program?"

The programmer would have to give a table of values showing how long the program took for different sizes of input. For example, the table might be as shown in Display 18.15. This table does not give a single time, but instead gives different times for a variety of different input sizes. The table is a description of what is called a **function** in mathematics. Just as a (non-*void*) C++ function takes an argument and returns a value, so too does this function take an argument, which is an input size, and returns a number, which is the time the program takes on an input of that size. If we call this function T, then $T(10)$ is 2 seconds, $T(100)$ is 2.1 seconds, $T(1000)$ is 10 seconds, and $T(10{,}000)$ is 2.5 minutes. The table is just a sample of some of the values of this function T. The program will take some amount of time on inputs of every size. So although they are not shown in the table, there are also values for $T(1)$, $T(2)$, . . . , $T(101)$, $T(102)$, and so forth. For any positive integer N, $T(N)$ is the amount of time it takes for the program to sort N numbers. The function T is called the **running time** of the program.

DISPLAY 18.15 Some Values of a Running-Time Function

Input Size	Running Time
10 numbers	2 seconds
100 numbers	2.1 seconds
1000 numbers	10 seconds
10,000 numbers	2.5 minutes

So far we have been assuming that this sorting program will take the same amount of time on any list of N numbers. That need not be true. Perhaps it takes much less time if the list is already sorted or almost sorted. In that case, $T(N)$ is defined to be the time taken by the "hardest" list, that is, the time taken on that list of N numbers which makes the program run the longest. This is called the **worst-case running time.** In this chapter *we will always mean worst-case running time* when we give a running time for an algorithm or for some code.

The time taken by a program or algorithm is often given by a formula, such as $4N + 3$, $5N + 4$, or N^2. If the running time $T(N)$ is $5N + 5$, then on inputs of size N the program will run for $5N + 5$ time units.

Following is some code for searching an array `a` with N elements to determine whether a particular value `target` is in the array:

```
int i = 0;
bool found = false;
while (( i < N) && !(found))
  if (a[i] == target)
      found = true;
  else
      i++;
```

We want to compute some estimate of how long it will take a computer to execute this code. We would like an estimate that does not depend on which computer we use, either because we do not know which computer we will use or because we might use several different computers to run the program at different times. One possibility is to count the number of "steps," but it is not easy to decide what a step is. In this situation the normal thing to do is to count the number of **operations.** The term *operations* is almost as vague as the term *step*, but there is at least some agreement in practice about what qualifies as an operation. Let us say that, for this C++ code, each application of any of the following will count as an operation: =, <, &&, !, [], ==, and ++. The computer must do other things besides carry out these operations, but these seem to be the main things that it is doing and we will assume that they account for the bulk of the time needed to run this code. In fact, our analysis of time will assume that everything else takes no time at all and that the total time for our program to run is equal to the time needed to perform these operations. Although this is an idealization that clearly is not completely true, it turns out that this simplifying assumption works well in practice and so is often made when analyzing a program or algorithm.

Even with our simplifying assumption, we still must consider two cases: Either the value `target` is in the array or it is not. Let us first consider the case when `target` is not in the array. The number of operations performed will depend on the number of array elements searched. The operation = is performed two times before the loop is executed. Since we are assuming that `target` is not in the array, the loop will be executed N times, one for each element of the array. Each time the loop is executed, the following operations are performed: <, &&, !, [], ==, and ++ This adds six operators for each of N loop iterations. Finally, after N iterations, the Boolean expression is again checked and found to be false. This adds a final

three operations (<, &&, !).[2] If we tally all these operations, we get a total of $6N + 5$ operations when the `target` is not in the array. We will leave it as an exercise for you to confirm that if the target is in the array, then the number of operations will be $6N + 5$ *or less*. Thus, the worst-case running time is $T(N) = 6N + 5$ operations for any array of N elements and any value of `target`.

We just determined that the worst-case running time for our search code is $6N + 5$ operations. But operations is not a traditional unit of time, like nanoseconds, seconds, or minutes. If we want to know how long the algorithm will take on some particular computer, we must know how long it takes that computer to perform one operation. If an operation can be performed in 1 nanosecond, then the time will be $6N + 5$ nanoseconds. If an operation can be performed in 1 second, the time will be $6N + 5$ seconds. If we use a slow computer that takes 10 seconds to perform an operation, the time will be $60N + 50$ seconds. In general, if it takes the computer c nanoseconds to perform one operation, then the actual running time will be approximately $c(6N + 5)$ nanoseconds. (We say *approximately*, since we are making some simplifying assumptions and so the result may not be the absolutely exact running time.) This means that our running time of $6N + 5$ is a very crude estimate. To get the running time expressed in nanoseconds, you must multiply by some constant that depends on the particular computer you are using. Our estimate of $6N + 5$ is only accurate to "within a constant multiple." There is a standard notation for these sorts of estimates and we discuss this notation next.

Estimates on running time, such as the one we just went through, are normally expressed in something called **big-*O* notation.** (The O is the letter "Oh," not the digit zero.) Suppose we estimate the running time to be, say, $6N + 5$ operations and suppose we know that no matter what the exact running time of each different operation may turn out to be, there will always be some constant factor c such that the real running time is less than or equal to $c(6N + 5)$.

Under these circumstances, we say the code (or program or algorithm) runs in time $O(6N + 5)$. This is usually read as "big-O of $6N + 5$." We need not know what the constant c will be. In fact, it will undoubtedly be different for different computers, but we must know that there is one such c for any reasonable computer system. If the computer is very fast, then the c might be less than 1—say, 0.001. If the computer is very slow, the c might be very large—say, 1000. Moreover, since changing the units, say from nanosecond to second, only involves a constant multiple, there is no need to give any units of time.

Be sure to notice that a big-O estimate is an upper-bound estimate. We always approximate by taking numbers on the high side, rather than the low side, of the true count. Also notice that when performing a big-O estimate, we need not determine a very exact count of the number of operations performed. We only need an estimate that is correct "up to a constant multiple." If our estimate is twice as large as the true number, that is good enough.

---

[2] Because of short circuit evaluation, `!(found)` is not evaluated, so we actually get two, not three operations. However, the important thing is to obtain a good upper bound. If we add in one extra operation that is not significant.

An order of magnitude estimate, such as the previous $6N + 5$, contains a parameter for the size of the task solved by the algorithm (or program or piece of code). In our sample case, this parameter N was the number of array elements to be searched. Not surprisingly, it takes longer to search a larger number of array elements than it does to search a smaller number of array elements. Big-O running time estimates are always expressed as a function of the size of the problem. In this chapter all our algorithms will involve a range of values in some container. In all cases N will be the number of elements in that range.

The following is an alternative, pragmatic way to think about big-O estimates:

Look only at the term with the highest exponent and do not pay attention to constant multiples.

For example, all of the following are $O(N^2)$:

$$N^2 + 2N + 1,\ 3N^2 + 7,\ 100N^2 + N$$

All of the following are $O(N^3)$:

$$N^3 + 5N^2 + N + 1,\ 8N^3 + 7,\ 100N^3 + 4N + 1$$

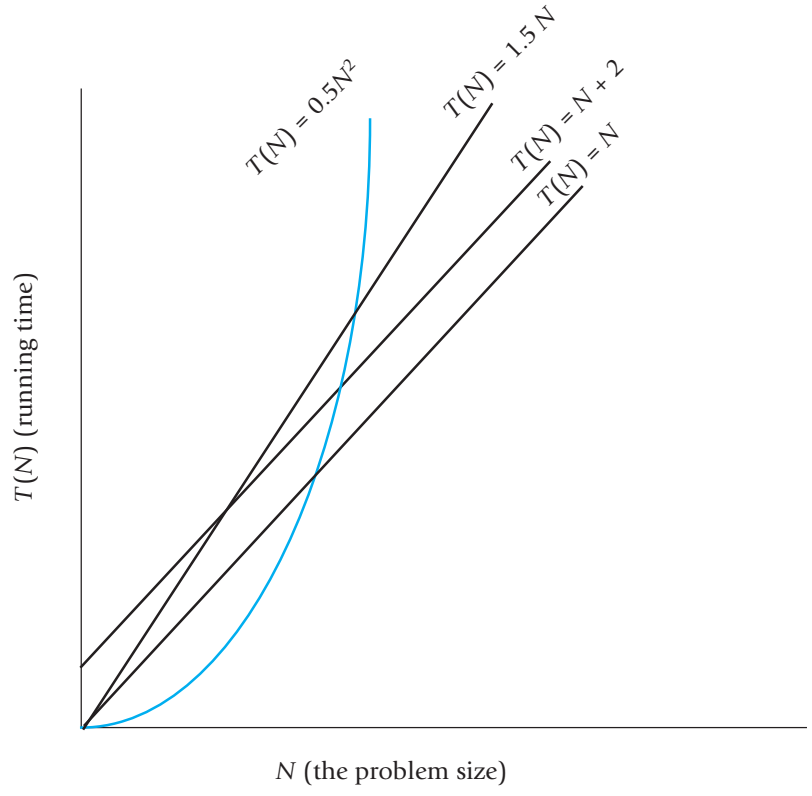
Big-O running-time estimates are admittedly crude, but they do contain some information. They will not distinguish between a running time of $5N + 5$ and a running time of $100N$, but they do let us distinguish between some running times and so determine that some algorithms are faster than others. Look at the graphs in Display 18.16; notice that all the graphs for functions that are $O(N)$ eventually fall below the graph for the function $0.5N^2$. The result is inevitable: An $O(N)$ algorithm will always run faster than any $O(N^2)$ algorithm, provided we use large enough values of N. Although an $O(N^2)$ algorithm could be faster than an $O(N)$ algorithm for the problem size you are handling, programmers have found that in practice $O(N)$ algorithms perform better than $O(N^2)$ algorithms for most practical applications that are intuitively "large." Similar remarks apply to any other two different big-O running times.

Some terminology will help with our descriptions of generic algorithm running times. **Linear running time** means a running time of $T(N) = aN + b$. A linear running time is always an $O(N)$ running time. **Quadratic running time** means a running time with highest term N^2. A quadratic running time is always an $O(N^2)$ running time. We will also occasionally have logarithms in running-time formulas. Those normally are given without any base, since changing the base is just a constant multiple. If you see $\log N$, think log base 2 of N, but it would not be wrong to think log base 10 of N. Logarithms are very slow-growing functions. So, a $O(\log N)$ running time is very fast. Sometimes $\log_2 N$ is written as $\lg N$.

Container Access Running Times

Now that we know about big-O notation, we can express the efficiency of some of the accessing functions for container classes that we discussed in Section 18.2 "Containers." Insertions at the back of a `vector` (`push_back`),

DISPLAY 18.16 Comparison of Running Times



the front or back of a `deque` (`push_back` and `push_front`), and anywhere in a `list` (`insert`) are all $O(1)$ (that is, a constant upper bound on the running time that is independent of the size of the container.) Insertion or deletion of an arbitrary element for a `vector` or `deque` is $O(N)$, where N is the number of elements in the container. For a `set` or `map`, finding (`find`) is $O(\log N)$, where N is the number of elements in the container.

SELF-TEST EXERCISES

17. Show that a running time $T(N) = aN + b$ is an $O(N)$ running time. (*Hint:* The only issue is the $+ b$. Assume N is always at least 1.)

18. Show that for any two bases a and b for logarithms, if a and b are both greater than 1, then there is a constant c such that $\log_a N \le c(\log_b N)$. Thus, there is no need to specify a base in $O(\log N)$. That is, $O(\log_a N)$ and $O(\log_b N)$ mean the same thing.

Nonmodifying Sequence Algorithms

This section describes template functions that operate on containers but do not modify the contents of the container in any way. A good simple and typical example is the generic `find` function.

The generic `find` function is similar to the `find` member function of the `set` template class but is a different `find` function; in particular, the generic `find` function takes more arguments than the `find` function we discussed when we presented the `set` template class. The generic `find` function searches a container to locate a particular element, but the generic `find` can be used with any of the STL sequential container classes. Display 18.17 shows a sample use of the generic `find` function used with the class `vector<char>`. The function in Display 18.17 would behave exactly the same if we replaced `vector<char>` with `list<char>` throughout, or if we replaced `vector<char>` with any other sequential container class. That is one of the reasons why the functions are called *generic*. One definition of the `find` function works for a wide selection of containers.

If the `find` function does not find the element it is looking for, it returns its second iterator argument, which need not be equal to some `end()` as it is in Display 18.17. Sample Dialogue 2 shows the situation when `find` does not find what it is looking for.

DISPLAY 18.17 The Generic `find` Function *(part 1 of 2)*

```
1   //Program to demonstrate use of the generic find function.
2   #include <iostream>
3   #include <vector>
4   #include <algorithm>
5   using std::cin;
6   using std::cout;
7   using std::endl;
8   using std::vector;
9   using std::find;

10  int main()
11  {
12      vector<char> line;

13      cout << "Enter a line of text:\n";
14      char next;
15      cin.get(next);
16      while (next != '\n')
17      {
18          line.push_back(next);
19          cin.get(next);
20      }
```

(continued)

DISPLAY 18.17 The Generic `find` Function *(part 2 of 2)*

```
21        vector<char>::const_iterator where;
22        where = find(line.begin(), line.end(), 'e');
23        //where is located at the first occurrence of 'e' in line.
24        vector<char>::const_iterator p;
25        cout << "You entered the following before you entered your first e:\n";
26        for (p = line.begin(); p != where; p++)
27            cout << *p;
28        cout << endl;
29        cout << "You entered the following after that:\n";
30        for (p = where; p != line.end(); p++)
31            cout << *p;
32        cout << endl;
33        cout << "End of demonstration.\n";
34        return 0;
35    }
```

If `find` *does not find what it is looking for, it returns its second argument.*

Sample Dialogue 1

```
Enter a line of text
A line of text.
You entered the following before you entered your first e:
A lin
You entered the following after that:
e of text.
End of demonstration.
```

Sample Dialogue 2

```
Enter a line of text
I will not!
You entered the following before you entered your first e:
I will not!
You entered the following after that:

End of demonstration.
```

If `find` *does not find what it is looking for, it returns* `line.end()`.

Does `find` work with absolutely any container classes? No, not quite. To start with, it takes iterators as arguments, and some containers, such as `stack`, do not have iterators. To use the `find` function, the container must have iterators, the elements must be stored in a linear sequence so that the

++ operator moves iterators through the container, and the elements must be comparable using ==. In other words, the container must have forward iterators (or some stronger kind of iterators, such as bidirectional iterators).

When presenting generic function templates, we will describe the iterator type parameter by using the name of the required kind of iterator as the type parameter name. So `ForwardIterator` should be replaced by a type that is a type for some kind of forward iterator, such as the `iterator` type in a `list`, `vector`, or other container template class. Remember, a bidirectional iterator is also a forward iterator, and a random access iterator is also a bidirectional iterator. So the type name `ForwardIterator` can be used with any iterator type that is a bidirectional or random access iterator type as well as a plain old forward iterator type. In some cases, when we specify `ForwardIterator` you can use an even simpler iterator kind; namely, an input iterator or output iterator, but since we have not discussed input and output iterators, we do not mention them in our function template declarations.

Remember the names *forward iterator*, *bidirectional iterator*, and *random access iterator* refer to kinds of iterators, not type names. The actual type names will be something like `std::vector<int>::iterator`, which in this case happens to be a random access iterator.

Display 18.18 gives a sample of some nonmodifying generic functions in the STL. The display uses a notation that is common when discussing container iterators. The iterator locations encountered in moving from an iterator `first` to, but not equal to, an iterator `last` is called the **range [first, last)**. For example, the following *for* loop outputs all the elements in the range `[first, last)`:

```
for (iterator p = first; p != last; p++)
    cout << *p << endl;
```

Note that when two ranges are given they need not be in the same container or even in the same type of container. For example, for the `search` function, the ranges `[first1, last1)` and `[first2, last2)` may be in the same or different containers.

Range `[first, last)`

The movement from some iterator `first`, often `container.begin()`, up to but not including some location `last`, often `container.end()`, is so common it has come to have a special name, **range** `[first, last)`. For example, the following outputs all elements in the range `[c.begin(),c.end())`, where `c` is some container object, such as a vector:

```
for (iterator p = c.begin(); p != c.end(); p++)
    cout << *p << endl;
```

DISPLAY 18.18 Some Nonmodifying Generic Functions

These all work for forward iterators, which means they also work for bidirectional and random access iterators. (In some cases they even work for other kinds of iterators, which we have not covered in any detail.)

```
1   template<class ForwardIterator, class T>
2   ForwardIterator find(ForwardIterator first,
3                        ForwardIterator last, const T& target);
4   //Traverses the range [first, last) and returns an iterator located at
5   //the first occurrence of target. Returns second if target is not found.
6   //Time complexity: linear in the size of the range [first, last).

7   template<class ForwardIterator, class T>
8   int[3] count(ForwardIterator first, ForwardIterator last, const T& target);
9   //Traverses the range [first, last) and returns the number
10  //of elements equal to target.
11  //Time complexity: linear in the size of the range [first, last).

12  template<class ForwardIterator1, class ForwardIterator2>
13  bool equal(ForwardIterator1 first1, ForwardIterator1 last1,
14             ForwardIterator2 first2);
15  //Returns true if [first1, last1) contains the same elements in the same order as
16  //the first last1-first1 elements starting at first2. Otherwise, returns false.
17  //Time complexity: linear in the size of the range [first, last).
18
19  template<class ForwardIterator1, class ForwardIterator2>
20  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
21                          ForwardIterator2 first2, ForwardIterator2 last2);
22  //Checks to see if [first2, last2) is a subrange of [first1, last1).
23  //If so, it returns an iterator located in [first1, last1) at the start of
24  //the first match. Returns last1 if a match is not found.
25  //Time complexity: quadratic in the size of the range [first1, last1).

26  template<class ForwardIterator, class T>
27  bool binary_search(ForwardIterator first, ForwardIterator last,
28                     const T& target);
29  //Precondition: The range [first, last) is sorted into ascending order using <.
30  //Uses the binary search algorithm to determine if target is in the range
31  //[first, last).
32  //Time complexity: For random access iterators O(log N). For non-random-access
33  //iterators
34  //linear is N, where N is the size of the range [first, last).
```

[3] The actual return type is an integer type that we have not discussed, but the returned value should be assignable to a variable of type int.

Notice that there are three search functions in Display 18.18—`find`, `search`, and `binary_search`. The function `search` searches for a subsequence, while the `find` and `binary_search` functions search for a single value. How do you decide whether to use `find` or `binary_search` when searching for a single element? One returns an iterator and the other returns just a Boolean value, but that is not the biggest difference. The `binary_search` function requires that the range being searched be sorted (into ascending order using `<`) and run in time $O(\log N)$; the `find` function does not require that the range be sorted but it guarantees only linear time. If you have or can have the elements in sorted order, you can search for them much more quickly by using `binary_search`.

Note that with the `binary_search` function you are guaranteed that the implementation will use the binary search algorithm, which was discussed in Chapter 14. The importance of using the binary search algorithm is that it guarantees a very fast running time, $O(\log N)$. If you have not read Chapter 14 and have not otherwise heard of binary search, just think of it as a very efficient search algorithm that requires that the elements be sorted. Those are the only two points about binary search that are relevant to the material in this chapter.

SELF-TEST EXERCISES

19. Replace all occurrences of the identifier `vector` with the identifier `list` in Display 18.17. Compile and run the program.

20. Suppose `v` is an object of the class `vector<int>`. Use the `search` generic function (Display 18.18) to write some code to determine whether or not `v` contains the number `42` immediately followed by `43`. You need not give a complete program, but do give all necessary `include` and `using` directives. (*Hint:* It may help to use a second vector.)

Container Modifying Algorithms

Display 18.19 contains descriptions of some of the generic functions in the STL which change the contents of a container in some way.

Remember that when you add or remove an element to or from a container, that can affect any of the other iterators. There is no guarantee that the iterators will be located at the same element after an addition or deletion unless the container template class makes such a guarantee. Of the template classes we have seen, `list` and `slist` guarantee that their iterators will not be moved by additions or deletions, except of course if the iterator is located at an element that is removed. The template classes `vector` and `deque` make no such guarantee. Some of the function templates in Display 18.19 guarantee the values of some specific iterators and those guarantees you can, of course, count on, no matter what the container is.

DISPLAY 18.19 Some Modifying Generic Functions

```
template<class T>
void swap(T& variable1, T& variable2);
//Interchanges the values of variable1 and variable2
```

The name of the iterator type parameter tells the kind of iterator for which the function works. Remember that these are minimum iterator requirements. For example, `ForwardIterator` *works for forward iterators, bidirectional iterators, and random access iterators.*

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
//Precondition: The ranges [first1, last1) and [first2, last2) are the same size.
//Action: Copies the elements at locations [first1, last1) to locations
//[first2, last2).
//Returns last2.
//Time complexity: linear in the size of the range [first1, last1).

template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& target);
//Removes those elements equal to target from the range [first, last).
//The size of
//the container is not changed. The removed values equal to target are
//moved to the
//end of the range [first, last). There is then an iterator i in this
//range such that
//all the values not equal to target are in [first, i). This i is returned.
//Time complexity: linear in the size of the range [first, last).

template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
//Reverses the order of the elements in the range [first, last).
//Time complexity: linear in the size of the range [first, last).

template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
//Uses a pseudorandom number generator to randomly reorder the elements
//in the range [first, last).
//Time complexity: linear in the size of the range [first, last).
```

SELF-TEST EXERCISES

21. Can you use the `random_shuffle` template function with a `list` container?

22. Can you use the `copy` template function with `vector` containers, even though `copy` requires forward iterators and `vector` has random access iterators?

Set Algorithms

Display 18.20 shows a sample of the generic `set` operation functions defined in the STL. Note that these generic algorithms assume the containers store their elements in sorted order. The containers `set`, `map`, `multiset`, and `multimap` do store their elements in sorted order, so all the functions in Display 18.20 apply to these four template class containers. Other containers, such as `vector`, do not store their elements in sorted order and these functions should not be used with such containers. The reason for requiring that the elements be sorted is so that the algorithms can be more efficient.

DISPLAY 18.20 Set Operations *(part 1 of 2)*

These operations work for sets, maps, multisets, multimaps (and other containers) but do not work for all containers. For example, they do not work for vectors, lists, or deques unless their contents are sorted. For these containers to work, the elements in the container must be stored in sorted order. These operators all work for forward iterators, which means they also work for bidirectional and random access iterators. (In some cases they even work for other kinds of iterators, which we have not covered in any detail.)

```
1   template<class ForwardIterator1, class ForwardIterator2>
2   bool includes(ForwardIterator1 first1, ForwardIterator1 last1,
3                 ForwardIterator2 first2, ForwardIterator2 last2);
4   //Returns true if every element in the range [first2, last2) also occurs in the
5   //range [first1, last1). Otherwise, returns false.
6   //Time complexity: linear in the size of [first1, last1) plus [first2, last2).
7
8   template<class ForwardIterator1, class ForwardIterator2,
9            class ForwardIterator3>
10  void set_union(ForwardIterator1 first1, ForwardIterator1 last1,
11                 ForwardIterator2 first2, ForwardIterator2 last2,
12                                          ForwardIterator3 result);
13  //Creates a sorted union of the two ranges [first1, last1) and [first2, last2).
14  //The union is stored starting at result.
15  //Time complexity: linear in the size of [first1, last1) plus [first2, last2).

16  template<class ForwardIterator1, class ForwardIterator2,
17           class ForwardIterator3>
18  void set_intersection(ForwardIterator1 first1, ForwardIterator1 last1,
19                        ForwardIterator2 first2, ForwardIterator2 last2,
20                                                 ForwardIterator3 result);
21  //Creates a sorted intersection of the two ranges [first1, last1) and
22  //[first2, last2).
23  //The intersection is stored starting at result.
24  //Time complexity: linear in the size of [first1, last1) plus [first2, last2).
25
26  template<class ForwardIterator1, class ForwardIterator2,
27           class ForwardIterator3>
```

(continued)

DISPLAY 18.20 Set Operations *(part 2 of 2)*

```
28    void set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
29                  ForwardIterator2 first2, ForwardIterator2 last2,
30                                           ForwardIterator3 result);
31    //Creates a sorted set difference of the two ranges [first1, last1) and
32    //[first2, last2).
33    //The difference consists of the elements in the first range that are not in the
34    //second.
35    //The result is stored starting at result.
36    //Time complexity: linear in the size of [first1, last1) plus [first2, last2).
```

SELF-TEST EXERCISE

23. The mathematics course version of a set does not keep its elements in sorted order and it has a union operator. Why does the `set_union` template function require that the containers keep their elements in sorted order?

Sorting Algorithms

Display 18.21 gives the declarations and documentation for two template functions, one to sort a range of elements and one to merge two sorted ranges of elements. Note that the sorting function `sort` guarantees a run time of $O(N \log N)$. Although it is beyond the scope of this book, it can be shown that you cannot write a comparison-based sorting algorithm that is faster than $O(N \log N)$. So this guarantees that the sorting algorithm is as fast as is possible, up to a constant multiple.

DISPLAY 18.21 Some Generic Sorting Algorithms

```
1     template<class RandomAccessIterator>
2     void sort(RandomAccessIterator first, RandomAccessIterator last);
3     //Sorts the elements in the range [first, last) into ascending order.
4     //Time complexity: O(N log N), where N is the size of the range [first, last).
5
6     template<class ForwardIterator1,class ForwardIterator2,
7              class ForwardIterator3>
8     void merge(ForwardIterator1 first1, ForwardIterator1 last1,
9                         ForwardIterator2 first2, ForwardIterator2 last2,
10                        ForwardIterator3 result);
11    //Precondition: The ranges [first1, last1) and [first2, last2) are sorted.
12    //Action: Merges the two ranges into a sorted range [result, last3), where
13    //last3 = result + (last1 - first1) + (last2 - first2).
14    //Time complexity: linear in the size of the range [first1, last1)
15    //plus the size of [first2, last2).
```

Sorting uses the < operator, and so the < operator must be defined. There are other versions, not given here, that allow you to provide the ordering relation. Sorted means sorted into ascending order.

CHAPTER SUMMARY

- An iterator is a generalization of a pointer. Iterators are used to move through the elements in some range of a container. The operations ++, --, and dereferencing * are usually defined for an iterator.
- Container classes with iterators have member functions `end()` and `begin()` that return iterator values such that you can process all the data in the container as follows:

```
for (p = c.begin(); p != c.end(); p++)
    process *p //*p is the current data item.
```

- The main kinds of iterators are

  Forward iterators: ++ works on the iterator.

  Bidirectional iterators: both ++ and -- work on the iterator.

  Random access iterators: ++, --, and random access all work with the iterator.

- With a constant iterator `p`, the dereferencing operator `*p` produces a read-only version of the element. With a mutable iterator `p`, `*p` can be assigned a value.
- A bidirectional container has reverse iterators that allow your code to cycle through the elements in the container in reverse order.
- The main container template classes in the STL are `list`, which has mutable bidirectional iterators, and the template classes `vector` and `deque`, both of which have mutable random access iterators.
- `stack` and `queue` are container adaptor classes, which means they are built on top of other container classes. A `stack` is a last-in/first-out container. A `queue` is a first-in/first-out container.
- The `set`, `map`, `multiset`, and `multimap` container template classes store their elements in sorted order for efficiency of search algorithms. A `set` is a simple collection of elements. A `map` allows storing and retrieving by key values. The `multiset` class allows repetitions of entries. The `multimap` class allows a single key to be associated with multiple data items.
- The STL includes template functions to implement generic algorithms with guarantees on their maximum running time.

Answers to Self-Test Exercises

1. `v.begin()` returns an iterator located at the first element of `v`. `v.end()` returns a value that serves as a sentinel value at the end of all the elements of `v`.

2. `*p` is the dereferencing operator applied to `p`. `*p` is a reference to the element at location `p`.

3.
```
vector<int>::iterator p;

for (p = v.begin(), p++; p != v.end(); p++)
    cout << *p << " ";
```

4. `D C C`

5. `B C`

6. Either would work.

7. A major difference is that a `vector` container has random access iterators whereas a `list` has only bidirectional iterators.

8. All except `slist`.

9. `vector` and `deque`.

10. They all can have mutable iterators.

11. The `stack` template adapter class has no iterators.

12. The `queue` template adapter class has no iterators.

13. No value is returned; `pop` is a `void` function.

14. `mymap` will contain two entries. One is a mapping from 5 to `"c++"` and the other is a mapping from 4 to the default string, which is blank.

15. Yes they can be of any type, although there is only one type for each `set` object. The type parameter in the template class is the type of elements stored.

16. If `'A'` is in `s`, then `s.find('A')` returns an iterator located at the element `'A'`. If `'A'` is not in `s`, then `s.find('A')` returns `s.end()`.

17. Just note that $aN + b \leq (a + b)N$, as long as $1 \leq N$.

18. This is mathematics, not C++, so = will mean *equals* not assignment.

    First note that $\log_a N = (\log_a b)(\log_b N)$.

    To see this first identity, just note that if you raise a to the power $\log_a N$, you get N, and if you raise a to the power $(\log_a b)(\log_b N)$, you also get N.

    If you set $c = (\log_a b)$, you get $\log_a N = c(\log_b N)$.

19. The programs should run exactly the same.

20.
```
#include <iostream>
#include <vector>
#include <algorithm>
using std::cout;
using std::vector;
using std::search;
```

```
...
vector<int> target;
target.push_back(42);
target.push_back(43);
vector<int>::const_iterator result = search(v.begin(), v.end(),
                        target.begin(), target.end());
if (result != v.end())
    cout << "Found 42, 43.\n";
else
    cout << "42, 43 not there.\n";
```

21. No, you must have random access iterators, and the `list` template class has only bidirectional iterators.

22. Yes, a random access iterator is also a forward iterator.

23. The `set_union` template function requires that the containers keep their elements in sorted order to allow the function template to be implemented in a more efficient way.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a program in which you declare a `deque` to store values of type *`double`*, read in ten *`double`* numbers, and store them in the `deque`. Then call the generic `sort` function to sort the numbers in the `deque` and display the results.

2. Write a program that uses the `map` template class to compute a histogram of positive numbers entered by the user. The `map`'s key should be the number that is entered, and the value should be a counter of the number of times the key has been entered so far. Use –1 as a sentinel value to signal the end of user input. For example, if the user inputs:

VideoNote
Solution to Practice Program 18.2

```
5
12
3
5
5
3
21
-1
```

then the program should output the following (not necessarily in this order):

```
The number 3 occurs 2 times.
The number 5 occurs 3 times.
The number 12 occurs 1 times.
The number 21 occurs 1 times.
```

3. Given a variable of type `string` set to arbitrary text, write a program that uses the `stack` template class of type `char` to reverse the string.

4. You have a list of student ID's followed by the course number (separated by a space) that the student is enrolled in. The listing is in no particular order. For example, if student 1 is in CS100 and CS200 while student 2 is in CS105 and MATH210 then the list might look like this:

   ```
   1 CS100
   2 MATH210
   2 CS105
   1 CS200
   ```

   Write a program that reads data in this format from the console. If the ID is -1 then stop inputting data. Use the `map` template class to map from an integer (the student ID) to a `vector` of type `string` that holds each class that the student is enrolled in.

   After all data is input, iterate through the map and output the student ID and all classes stored in the vector for that student. The result should be a list of classes organized by student ID.

   If you aren't using C++11 or higher then don't forget that you need a space between the >> characters when defining the map of vectors.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a program that allows the user to enter any number of student names and their scores. The program should then display the student names and scores according to the ascending order of scores. Use the template class `vector` and the generic `sort` function from the STL. Note that you will need to define a structure or class type for data consisting of one student name and score. You will also need to overload the < operator for this structure or class.

2. A **prime** number is an integer greater than 1 and divisible only by itself and 1. An integer `x` is **divisible** by an integer `y` if there is another integer `z` such that `x = y * z`. The Greek mathematician Eratosthenes (pronounced: Er-ah-**tos**-thin-eeze) gave an algorithm, called the *Sieve of Eratosthenes*, for finding all prime numbers less than some integer *N*. The algorithm works like this: Begin with a list of integers 2 through *N*. The number 2 is the first prime. (It is instructive to consider why this is true.) The *multiples* of 2, that is, 4, 6, 8, etc., are *not prime*. We cross these off the list. Then the first number after 2 that was not crossed off is the next prime. This number is 3. The *multiples*

of 3 are not primes. Cross the multiples of 3 off the list. Note that 6 is already gone, cross off 9, 12 is already gone, cross off 15, etc. The first number not crossed off is the next prime. The algorithm continues on in this fashion until we reach *N*. All the numbers not crossed off the list are primes.

a. Write a program using this algorithm to find all primes less than a user-supplied number *N*. Use a vector container for the integers. Use an array of `bool` initially set to all `true` to keep track of crossed-off integers. Change the entry to `false` for integers that are crossed off the list.

b. Test for *N* = 10, 30, 100, and 300.

We can improve our solution in several ways:

c. The program does not need to go all the way to *N*. It can stop at *N*/2. Try this and test your program. *N*/2 works and is better but is not the smallest number we could use. Argue that to get all the primes between 1 and *N* the minimum limit is the square root of *N*.

d. Modify your code from part (a) to use the square root of *N* as an upper limit.

3. Suppose you have a collection of student records. The records are structures of the following type:

```
struct StudentInfo
{
    string name;
    int grade;
};
```

The records are maintained in a `vector<StudentInfo>`. Write a program that prompts for and fetches data and builds a vector of student records, then sorts the vector by name, calculates the maximum and minimum grades and the class average, then prints this summarizing data along with a class roll with grades. (We aren't interested in who had the maximum and minimum grade, though, just the maximum, minimum, and average statistics.) Test your program.

4. Continuing Programming Project 3, write a function that separates the students in the vector of `StudentInfo` records into two vectors, one containing records of passing students and one containing records of failing students. (Use a grade of 60 or better for passing.)

You are asked to do this in two ways, and to give some run-time estimates.

a. Consider continuing to use a vector. You could generate a second vector of passing students and a third vector of failing students. This keeps duplicate records for at least some of the time, so don't do it that way. You could create a vector of failing students and a test-for-failing function. Then you `push_back` failing student records, then `erase`

(which is a member function) the failing student records from the original vector. Write the program this way.

b. Consider the efficiency of this solution. You are potentially erasing *O*(*N*) members from the middle of a vector. You have to move a lot of members in this case. `erase` from the middle of a vector is an *O*(*N*) operation. Give a big-*O* estimate of the running time for this program.

c. If you used a `list<StudentInfo>`, what are the run-times for the `erase` and `insert` functions? Consider how the time efficiency of `erase` for a `list` affects the run-time for the program. Rewrite this program using a `list` instead of a `vector`. Remember that a `list` provides neither indexing nor random access and its iterators are only bidirectional, not random access.

5. Redo (or do for the first time) Programming Project 9 from Chapter 11, except use the STL `set` template class instead of your own `set` class. Use the generic `set_intersection` function to compute the intersection of Q and D.

   Here is an example of `set_intersection` to intersect set A with B and store the result in C, where all sets are sets of strings:

```
#include <iterator>
#include <set>
#include <string>
...

set<string> C;
// Note space between >> in line below
insert_iterator<set<string> > cIterator(C, C.begin());
set_intersection(A.begin(), A.end(),
                 B.begin(),B.end(),
                 cIter);
// set C now contains the intersection of A and B
```

VideoNote
Solution to Programming Project 18.6

6. In this project you are to create a database of books that are stored using a vector. Keep track of the author, title, and publication date of each book. Your program should have a main menu that allows the user to select from the following: (1) Add a book's author, title, and date; (2) Print an alphabetical list of the books sorted by author; and (3) Quit.

   You must use a class to hold the data for each book. This class must hold three string fields: one to hold the author's name, one for the publication date, and another to hold the book's title. Store the entire database of books in a vector in which each vector element is a book class object.

   To sort the data, use the generic `sort` function from the `<algorithm>` library. Note that this requires you to define the < operator to compare two objects of type Book so that the author field from the two books are compared.

A sample of the input/output behavior might look as follows. Your I/O need not look identical, this is just to give you an idea of the functionality.

```
Select from the following choices:
1.          Add new book
2.          Print listing sorted by author
3.          Quit
1

Enter title:
More Than Human

Enter author:
Sturgeon, Theodore

Enter date:
1953

Select from the following choices:
1.          Add new book
2.          Print listing sorted by author
3.          Quit
1

Enter title:
Problem Solving with C++

Enter author:
Savitch, Walter

Enter date:
2015

Select from the following choices:
1.          Add new book
2.          Print listing sorted by author
3.          Quit
2

The books entered so far, sorted alphabetically by author are:
            Savitch, Walter.  Problem Solving with C++.  2015.
            Sturgeon, Theodore.  More Than Human.  1953.

Select from the following choices:
1.         Add new book
2.         Print listing sorted by author
3.         Quit
1

Enter title:
At Home in the Universe

Enter author:
Kauffman

Enter date:
1996
```

```
Select from the following choices:
1.          Add new book
2.          Print listing sorted by author
3.          Quit
2

The books entered so far, sorted alphabetically by artist are:
             Kauffman, At Home in the Universe, 1996
             Savitch, Walter.  Problem Solving with C++.  2015.
             Sturgeon, Theodore.  More Than Human.  1953.
```

7. Redo or do for the first time Programming Project 8 from Chapter 14, except use the STL `set` class for all set operations and the STL `linked list` class to store and manipulate each individual permutation. When creating a set containing lists, make sure to place a space between the last two >'s if you are using a compiler earlier than C++11. For example, `set<list<int>>` defines a set where elements are linked lists containing elements of type `int`. The code `set<list<int> >` without a space will produce a compiler error. (This issue was eliminated with the release of C++11.)

8. You have collected a file of movie ratings where each movie is rated from 1 (bad) to 5 (excellent). The first line of the file is a number that identifies how many ratings are in the file. Each rating then consists of two lines: the name of the movie followed by the numeric rating from 1 to 5. Here is a sample rating file with four unique movies and seven ratings:

```
7
Harry Potter and the Order of the Phoenix
4
Harry Potter and the Order of the Phoenix
5
The Bourne Ultimatum
3
Harry Potter and the Order of the Phoenix
4
The Bourne Ultimatum
4
Wall-E
4
Glitter
1
```

   Write a program that reads a file in this format, calculates the average rating for each movie, and outputs the average along with the number of reviews. Here is the desired output for the sample data:

```
Glitter: 1 review, average of 1 / 5
Harry Potter and the Order of the Phoenix: 3 reviews, average
of 4.3 / 5
```

```
The Bourne Ultimatum: 2 reviews, average of 3.5 / 5
Wall-E: 1 review, average of 4 / 5
```

Use a map or multiple maps to calculate the output. Your map(s) should index from a string representing each movie's name to integers that store the number of reviews for the movie and the sum of the ratings for the movie.

9. Consider a text file of names, with one name per line, that has been compiled from several different sources. A sample follows:

```
Brooke Trout
Dinah Soars
Jed Dye
Brooke Trout
Jed Dye
Paige Turner
```

There are duplicate names in the file. We would like to generate an invitation list but don't want to send multiple invitations to the same person. Write a program that eliminates the duplicate names by using the `set` template class. Read each name from the file, add it to the `set`, and then output all names in the `set` to generate the invitation list without duplicates.

10. Do Programming Project 16 from Chapter 8 except use a `Racer` class to store information about each race participant. The class should store the racer's name, bib number, finishing position, and all of his or her split times as recorded by the RFID sensors. You can choose appropriate structures to store this information. Include appropriate functions to access or change the racer's information, along with a constructor.

Use a `map` to store the race data. The `map` should use the bib number as the key and the value should be the `Racer` object that corresponds to the bib number. With the map you won't need to search for a bib number anymore, you can directly access the splits and final position based on the bib number.

If you aren't using C++11 or higher then don't forget that you need a space between the >> characters when defining the map of vectors.