Standard Template Library 19

19.1 ITERATORS 859
Iterator Basics 859
Pitfall: Compiler Problems 864
Kinds of Iterators 865
Constant and Mutable Iterators 868
Reverse Iterators 870
Other Kinds of Iterators 871

19.2 CONTAINERS 872
Sequential Containers 872
Pitfall: Iterators and Removing Elements 877
Tip: Type Definitions in Containers 878
The Container Adapters `stack` and `queue` 878
Pitfall: Underlying Containers 879
The Associative Containers `set` and `map` 882
Efficiency 887

19.3 GENERIC ALGORITHMS 889
Running Times and Big-*O* Notation 890
Container Access Running Times 894
Nonmodifying Sequence Algorithms 895
Modifying Sequence Algorithms 899
Set Algorithms 900
Sorting Algorithms 902

Chapter Summary 902
Answers to Self-Test Exercises 903
Programming Projects 905

19 Standard Template Library

Libraries are not made; they grow.

AUGUSTINE BIRRELL

Introduction

In Chapter 17 we constructed our own versions of the stack and queue data structures. A large collection of standard structures for holding data exists. Because they are so standard, it makes sense to have standard portable implementations of these data structures. 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. Chapter 7 presented a preview of the STL by describing the `vector` template class, which is one of the container classes in the STL. This chapter presents an overview of some of the basic classes included in the STL. Because the STL is very large, we will not be able to give a comprehensive treatment of it here, but we will present enough to get you started using some basic STL container classes as well as some of the other items in the STL.

STL

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 includes the STL. As a practical matter, you can consider the STL to be part of the C++ language.

As its name suggests, 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 general concept of an iterator was given in Section 17.3 of Chapter 17. Although this chapter does not presuppose that you have read that section, most readers will find it helpful to read that section before reading this chapter. As defined in the STL, iterators are very general and can be used for more than just cycling through the few container classes we will cover. Our discussion of iterators will be specialized to simple uses with the container classes discussed in this chapter. This should make the concept come alive in a concrete setting and should give you enough understanding to feel comfortable reading more advanced texts on the STL (there are numerous books dedicated to the STL).

The STL also includes implementations of many important generic algorithms, such as searching and sorting. 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 that they are template classes and template functions.

If you have not already done so, you should read Section 7.3 of Chapter 7, which covers the `vector` template class of the STL. Although the current chapter does not use any of the material in Chapter 17, most readers will find that reading Chapter 17 before reading this one will aid his or her comprehension of this chapter by giving sample concrete implementations of some of the abstract ideas intrinsic to the STL. This chapter does not use any of the material in Chapters 12 to 15.

19.1 Iterators

To iterate is human, and programmers are human.

ANONYMOUS

If you have not yet done so, you should read Chapter 10 on pointers and arrays and also read Section 7.3 of Chapter 7, which covers vectors. Vectors are one of the container template classes in the STL. Iterators are a generalization of pointers. This section shows how to use iterators with vectors. Other container template classes that we introduce in Section 19.2 use iterators in the same way. So, all that you learn about iterators in this section will apply across a wide range of containers rather than applying 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 is) uniform across different container types.

Iterator Basics

iterator

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 type.

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. Like a pointer variable, an iterator variable is located at (meaning, it 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 or write only, or it may 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 when used to test whether a pointer has passed the last node in a linked list of the kind discussed in Chapter 17. The iterator c.end( ) is thus an iterator that is not located at a 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 is the big picture. Now let us look at the details in the concrete setting of the vector template container class.

Display 19.1 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 ints are of type

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

Another container class is the list template class. Iterators for lists of ints are of type

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

In the program in Display 19.1, we specialize the type name iterator so it applies to iterators for vectors of ints. The type name iterator that we want in Display 19.1 is defined in the template class vector. Thus, if we specialize the template class vector to ints and want the iterator type for vector<int>, we want the type

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

Display 19.1 Iterators Used with a Vector

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<int> container;

    for (int i = 1; i <= 4; i++)
        container.push_back(i);

    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

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

    cout << "Container now contains:\n";
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

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

The basic use of iterators with `vector` (or any container class) is illustrated by the following lines from Display 19.1:

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

Recall that `container` is of type `vector<int>`, and that the type `iterator` really means `std::vector<int>::iterator`.

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 (or points 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 by 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 comparison 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.

begin( )

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. The basic `for` loop for visiting all elements of the vector `v` is therefore

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

The desired stopping condition is

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

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 correct `Boolean_Expression` is the negation of this stopping condition:

```
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. It is not an iterator, but you can compare `v.end( )` to an iterator using `==` and `!=`. The value `v.end( )` is analogous to the value `NULL` that is used to mark the end of a linked list of the kind discussed in Chapter 17.

The following `for` loop from Display 19.1 uses this same 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`. The preceding `cout` statement thus outputs the element located at the iterator `p`, and so 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 it. For vectors, `*p` will allow you to change the element located at `p`, as illustrated by the following `for` loop from Display 19.1:

```
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`.

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;
```

Alternatively, you could use the following:

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

You could also use the following, which is not quite as nice:

```
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 these alternatives. If one form does not work with your compiler, try another. ■

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. 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 as follows:

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

`STL_container` is the name of the container class (e.g., `vector`) and `datatype` is the data type of items to be stored in the container. 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.

Dereferencing

The dereferencing operator, `*p`, when applied to an iterator `p`, 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.

MyProgrammingLab™

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 will output all the elements of `p` 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. Thus, we will again use the vector container to illustrate iterators. In this case we use a vector to illustrate the iterator operations of *decrement* and *random access*. Display 19.2 shows another program using a vector object named `container` and an iterator `p`.

The decrement operator is used on line 29 of Display 19.2. As you would expect, `p--` moves the iterator `p` to the previous location. The decrement operator, `--`, is similar to 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 then 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; we use them only to change the variable value.

The following lines from Display 19.2 illustrate the fact 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

Random access means that you can go directly to any particular element in one step. We have already used `container[2]` as a form of random access to a vector. This 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 10), `(p + 2)` names the location two places beyond `p`. Since `p` is at the first (index `0`) location in the previous 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 to a different element.

Display 19.2 **Bidirectional and Random-Access Iterator Use** (part 1 of 2)

```
//Program to demonstrate bidirectional and random-access iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == "
             << container[i] << endl;

    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;

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

    cout << "Two steps forward and one step back:\n";
    p++;
    cout << *p << endl;

    p++;
    cout << *p << endl;
    p--;
    cout << *p << endl;

    return 0;
}
```

Three different notations for the same thing.

This notation is specialized to vectors and arrays.

These two work for any random-access iterator.

This works for any bidirectional iterator.

Display 19.2 **Bidirectional and Random-Access Iterator Use** (part 2 of 2)

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
```

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. Something similar happens with `p[2]` behind the scenes. 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 19.2 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]`.

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

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. An actual type name would be something like `std::vector<int>::iterator`, which in this case happens to be a random-access iterator.

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.

MyProgrammingLab™

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

constant iterator

The categories of 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` to assign it to a variable or output it to the screen, for example, but you cannot change the element in the container by, for example, assigning mutable iterator

to *p. With a **mutable iterator** p, *p can be assigned a value, which will change the corresponding element in the container. Phrased another way, with a mutable iterator p, *p returns an lvalue. The vector iterators are mutable, as shown by the following lines from Display 19.1:

```
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 device if you intend that your code should 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 19.2 would behave exactly the same if you replaced

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

with

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

However, a similar change would not work in Display 19.1 because of the following line from the program in Display 19.1:

```
*p = 0;
```

Note that const_iterator is a type name, whereas *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 the following:

```
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 it: `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:

reverse iterator

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

rbegin( )
rend( )

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 19.3 demonstrates a reverse iterator.

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

Reverse Iterators

A *reverse iterator* can be used to cycle through all elements of a container with bidirectional iterators. The elements are visited in reverse order. The general scheme is as follows:

```
STL_container<datatype>::reverse_iterator rp;
for (rp = c.rbegin( ); rp != c.rend( ); rp++)
    Process_At_Location p;
```

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

When using `reverse_iterator`, you need to have some sort of `using` declaration or something equivalent. For example, if `c` is a `vector<int>`, the following will suffice:

```
vector<int>::reverse_iterator rp;
```

Display 19.3 **Reverse Iterator**

```
//Program to demonstrate a reverse iterator.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');

    cout << "Forward:\n";
    vector<char>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Reverse:\n";
    vector<char>::reverse_iterator rp;
    for (rp = container.rbegin( ); rp != container.rend( ); rp++)
        cout << *rp << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

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

Other Kinds of Iterators

input iterator

output iterator

There are other kinds of iterators, which we will not cover in this book. We will briefly mention two kinds of iterators whose names you may encounter. An **input iterator** is essentially a forward iterator that can be used with input streams. An **output iterator** is essentially a forward iterator that can be used with output streams. For more details you will need to consult a more advanced reference.

MyProgrammingLab™

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;
```

19.2 Containers

You can put all your eggs in one basket, but be sure it's a good basket.

WALTER SAVITCH, *Absolute C++*

container class

The **container classes** of the STL are different kinds of 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, whereas 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.

Sequential Containers

singly linked list

A sequential container arranges its data items into a list such that there is a first element, a next element, and so forth, up to a last element. The linked lists we discussed in Chapter 17 are examples of a kind of sequential container; these kinds of lists 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 a singly linked list, although some implementations do offer an implementation of a singly linked list, typically under

doubly linked list

the name `slist`.[1] 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 19.4 and is described in more detail in Section 17.1.

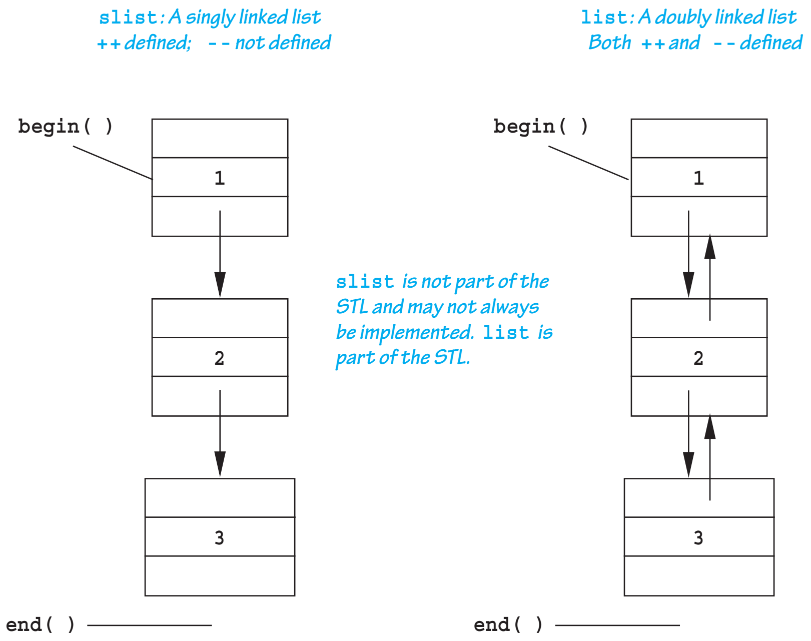
`slist` and `list`

The lists in Display 19.4 contain the three integer values `1`, `2`, and `3` in that order. The types for the two lists are `slist<int>` and `list<int>`. The 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 19.4 we have drawn our singly and doubly linked lists as nodes and pointers of the form discussed in Chapter 17. 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 of iterators (not pointers). You can think of the arrows in Display 19.4 as indicating the directions for ++ (which is down) and -- (which is up).

We presented the template class `slist` to help give a context for the sequential containers. It corresponds to what we discussed in Chapter 17 and is the first thing

Display 19.4 **Two Kinds of Lists**


slist: A singly linked list
++ defined; -- not defined
list: A doubly linked list
Both ++ and -- defined
begin( )
1
2
3
end( )
begin( )
1
2
3
end( )
slist is not part of the STL and may not always be implemented. list is part of the STL.

[1]The Silicon Graphics version of the STL includes `slist` and is distributed with the g++ compiler. SGI provides a very useful reference document for its STL version that is applicable to almost everyone's STL.

that comes to the mind of most programmers when you mention *linked lists*. However, since the template class `slist` is not standard, we will not discuss it further. 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`.

push_back

A simple program using the STL template class `list` is given in Display 19.5. 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 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.

Note that Display 19.5 would compile and run exactly the same if we replaced `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, whereas a list has only bidirectional iterators. For example, if you start with Display 19.2, 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]`.)

memory management

The basic sequential container template classes of the STL are listed in Display 19.6. Other containers, such as stacks and queues, can be obtained from these using techniques discussed in the subsection entitled "The Container Adapters `stack` and `queue`." A sample of some member functions of the sequential container classes is given in Display 19.7. All these sequence template classes have a destructor that returns storage for recycling.

deque

Deque is pronounced "d-queue" or "deck" and 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, 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`.

Display 19.5 Using the `list` Template Class

```
//Program to demonstrate the STL template class list.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main( )
{
    list<int> listObject;

    for (int i = 1; i <= 3; i++)
        listObject.push_back(i);

    cout << "List contains:\n"
    list<int>::iterator iter;
    for (iter = listObject.begin( ); iter != listObject.end( );
                                     iter++)
        cout << *iter << " ";
    cout << endl;

    cout << "Setting all entries to 0:\n"
    for (iter = listObject.begin( ); iter != listObject.end( );
                                     iter++)
        *iter = 0;

    cout << "List now contains:\n"
    for (iter = listObject.begin( ); iter != listObject.end( );
                                     iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

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

Display 19.6 **STL Basic Sequential Containers**

TEMPLATE CLASS NAME	ITERATOR TYPE NAMES	KIND OF ITERATORS	LIBRARY HEADER FILE

`slist` (Warning: `slist` is not part of the STL.)	`slist<T>::iterator`
	`slist<T>::const_iterator`
		Mutable forward
		Constant forward
			`<slist>` 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 19.7 **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`.

Display 19.7 **Some Sequential Container Member Functions** (part 2 of 2)

`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*`)`	Inserts the *Element* at the end of the sequence. Not a member of `slist`.
`c.push_front(`*Element*`)`	Inserts the *Element* at the front of the sequence. Not a member of `vector`.
`c.insert(`*Iterator, Element*`)`	Inserts 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 sequence 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.	

PITFALL: Iterators and Removing Elements

Adding or removing an element to or from a container 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.

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 scene). There are typically other type definitions as well.

The type `value_type` is the type of the elements stored in the container, and `size_type` is an unsigned integer type that is the return type for the member function `size`. For example, `list<int>::value_type` is another name for `int`. All the template classes we have discussed so far have the defined types `value_type` and `size_type`. ■

MyProgrammingLab™

Self-Test Exercises

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

The 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 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.

priority queue

Other container adapter classes are the `queue` and `priority_queue` template classes. Stacks and queues were discussed in Chapter 17. A **priority queue** is 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 discuss 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 sequence container may serve as the underlying container for the `stack` template class, and any sequence container other than `vector` may serve as the underlying container for the `queue` template class. 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. Make sure to always insert a space between the two > symbols. We will always use the default underlying container.

Warning!

The member functions and other details about the `stack` template class are given in Display 19.8. The details for the `queue` template class are given in Display 19.9. A simple example of using the `stack` template class is given in Display 19.10.

PITFALL: Underlying Containers

If you specify an underlying container, be warned that you should not place two > symbols in the type expression without a space in between them, or the compiler can be confused. Use `stack<int, vector<int> >`, with a space between the last two >s. Do not use `stack<int, vector<int>>`. ■

MyProgrammingLab™

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()`?

Display 19.8 **The `stack` Template Class**

`stack` Adapter Template Class Details

Type name: `stack<T>` or `stack<T, Sequence_Type>` 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.

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, and a constructor that takes an object of any sequence 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 19.9 **The `queue` Template Class** (part 1 of 2)

`queue` Adapter Template Class Details

Type name: `queue<T>` or `queue<Sequence_Type, T>` for a queue of elements of type `T`. For efficiency reasons, the `Sequence_Type` 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.

Display 19.9 **The `queue` Template Class** (part 2 of 2)

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`.
`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, and a constructor that takes an object of any sequence 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 19.10 **Program Using the `stack` Template Class** (part 1 of 2)

```
//Program to demonstrate use of the stack template class from the STL.
#include <iostream>
#include <stack>
using std::cin;
using std::cout;
using std::endl;
using std::stack;

int main( )
{
    stack<char> s;
    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
```

(continued)

Display 19.10 **Program Using the `stack` Template Class** (part 2 of 2)

```
    while (next != '\n')
    {
        s.push(next);
        cin.get(next);
    }

    cout << "Written backward that is:\n";
    while ( ! s.empty( ) )
    {
        cout << s.top( );
        s.pop( );
    }
    cout << endl;

    return 0;
}
```

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

Sample Dialogue

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

The Associative Containers `set` and `map`

Associative containers are basically very simple databases. They store data, such as `struct`s 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 each data item is its own key. For example, in a `set`, every element is its own key.

key

set

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 that 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. 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.[2] `T` is the type of elements stored. If no ordering is specified, then the ordering is assumed to use the `<` relational operator. Some basic details about the `set` template class are given in Display 19.11. A simple example that shows how to use some of the member functions of the template class `set` is given in Display 19.12.

map

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> numberMap;
```

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

associative array

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 can also use the `insert` or `find` methods if you want.

Like a `set` object, a `map` object stores its elements sorted 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 are 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.

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. This 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`.

[2]The ordering must be a *strict weak ordering*. Most typical ordering used to implement the < operator is strict weak ordering. For those who want the details, a strict weak ordering must be one of the following: (irreflexive) Ordering(x, x) is always `false`; (antisymmetric) Ordering(x, y) implies !Ordering(y, x); (transitive) Ordering(x, y) and Ordering(y, z) implies 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 19.11 **The `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 an iterator located at the copy of *Element* in the set. If *Element* is not in the set, `s.end( )` is returned. Whether the iterator is mutable or not is implementation dependent.
`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 contain the same elements; otherwise, returns `false`.

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

Display 19.12 **Program Using the `set` Template Class** (part 1 of 2)

```
//Program to demonstrate use of the set template class.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;
```

Display 19.12 **Program Using the `set` Template Class** (part 2 of 2)

```
int main( )
{
    set<char> s;

    s.insert('A');
    s.insert('D');
    s.insert('D');
    s.insert('C');
    s.insert('C');
    s.insert('B');

    cout << "The set contains:\n";
    set<char>::const_iterator p;
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;

    cout << "Set contains 'C': ";
    if (s.find('C')==s.end( ))
       cout << " no " << endl;
    else
       cout << " yes " << endl;

    cout << "Removing C.\n";
    s.erase('C');
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;

    cout << "Set contains 'C': ";
    if (s.find('C')==s.end( ))
        cout << " no " << endl;
    else
        cout << " yes " << endl;

    return 0;
}
```

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
Set contains 'C': yes
Removing C.
A B D
Set contains 'C': no
```

Some basic details about the `map` template class are given in Display 19.13. In order to understand these details, you need to first know something about the `pair` template class.

`pair`

Display 19.13 The `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>`, which 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* above.)

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 can 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 *Target_Key*. If the map does not already contain such an object, then a default object of type `T` is inserted.
`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 contain the same pairs; otherwise, returns `false`.

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

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;
```

if we add to the map

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

then 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 19.14.

We will mention four 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 `multiset` allows repetition of elements and `multimap` allows multiple values to be associated with each key value. Some implementations of STL also include the `hash_set` and `hash_map` classes. These template classes are essentially the same as `set` and `map`, except they are implemented using a hash table. Hash tables are described in Chapter 17. Instead of hash tables, most implementations of the `set` and `map` classes use balanced binary trees. In a balanced binary tree, the number of nodes to the left of the root is approximately equal to the number of nodes to the right of the root. Binary search trees are also described in Chapter 17, although we do not discuss details of balancing them.

Efficiency

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 19.3. (Section 19.3 also gives some guaranteed running times for some of the container member functions we have already discussed. These are given in the subsection entitled "Container Access Running Times.") You will be told the guaranteed maximum running times for certain functions described in the rest of this chapter.

Display 19.14 **Program Using the `map` Template Class** (part 1 of 2)

```cpp
//Program to demonstrate use of the map template class.
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

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

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile-per-hour winds";
    planets["Pluto"] = "Dwarf planet";

    cout << "Entry for Mercury - " << planets["Mercury"]
            << endl << endl;

    if (planets.find("Mercury") != planets.end( ))
        cout << "Mercury is in the map." << endl;
    if (planets.find("Ceres") == planets.end( ))
        cout << "Ceres is not in the map." << endl << endl;

    cout << "Iterating through all planets: " << endl;
    map<string, string>::const_iterator iter;
    for (iter = planets.begin( ); iter != planets.end( ); iter++)
    {
        cout << iter->first << " - " << iter->second << endl;
    }
    return 0;
}
```

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

Sample Dialogue

```
Entry for Mercury - Hot planet

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

Display 19.14 **Program Using the `map` Template Class** (part 2 of 2)

```
Iterating through all planets:
Earth - Home
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
```

MyProgrammingLab™

Self-Test Exercises

14. Why are the elements in the `set` template class stored in sorted order?
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`?
17. How many elements will be in the map `mymap` after the following code executes?

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

19.3 Generic Algorithms

"And if you take one from three hundred and sixty-five, what remains?"
"Three hundred and sixty-four, of course."
Humpty Dumpty looked doubtful. "I'd rather see that done on paper," he said.

LEWIS CARROLL, *Through the Looking-Glass*

This section covers some basic function templates in the STL. We cannot give you a comprehensive description of them all here, but we 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.

generic algorithm

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, which 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, but also how rapidly the task will be done. In some cases, the standard even specifies the particular algorithm that is used, although not the exact details 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 the 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 for sorting numbers would take less time to sort 10 numbers than it would to sort 1000 numbers. Perhaps it takes 2 seconds to sort 10 numbers, but 10 seconds to sort 1000 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 takes for different sizes of input. For example, the table might be as shown in Display 19.15. This table does not give a single time, but instead gives different times for a variety of different input sizes.

mathematical function

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.

running time

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 that

worst-case running time

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.

The following is some code to search 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++;
```

Display 19.15 **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

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.

operations

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 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 it 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 five operations 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 (`<`, `&&`, `!`).[3] 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 the reader to confirm that if the target is in the array, then the number of operations will be $6N + 5$ *or fewer*. 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 an operation is not a traditional unit of time, like a nanosecond, second, or minute. 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 one nanosecond, then the time will be $6N + 5$ nanoseconds. If an operation can be performed in one second, the time will be $6N + 5$ seconds. If we use a slow computer that takes ten 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 said *approximately* because we are making some simplifying assumptions, and therefore 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.

big-*O* notation

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 that 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, 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) involves only 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

[3]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.

count. Also notice that when performing a big-O estimate, we need not determine an exact count of the number of operations performed. We need only 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.

size of task

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 $O(N^3)$:

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

These 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 19.16 and 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 $O(N)$ algorithms for most practical applications that are intuitively "large." Similar remarks apply to any other two different big-O running times.

linear running time

quadratic running time

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 a highest term of 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, an $O(\log N)$ running time is very fast.

In many cases, our running-time estimates will be better than big-O estimates. In particular, when we specify a linear running time, that is a tight upper bound; you can think of the running time as being exactly $T(N) = cN$, although the c is still not specified.

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 19.2. Insertions at the back of a `vector` (`push_back`), 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.

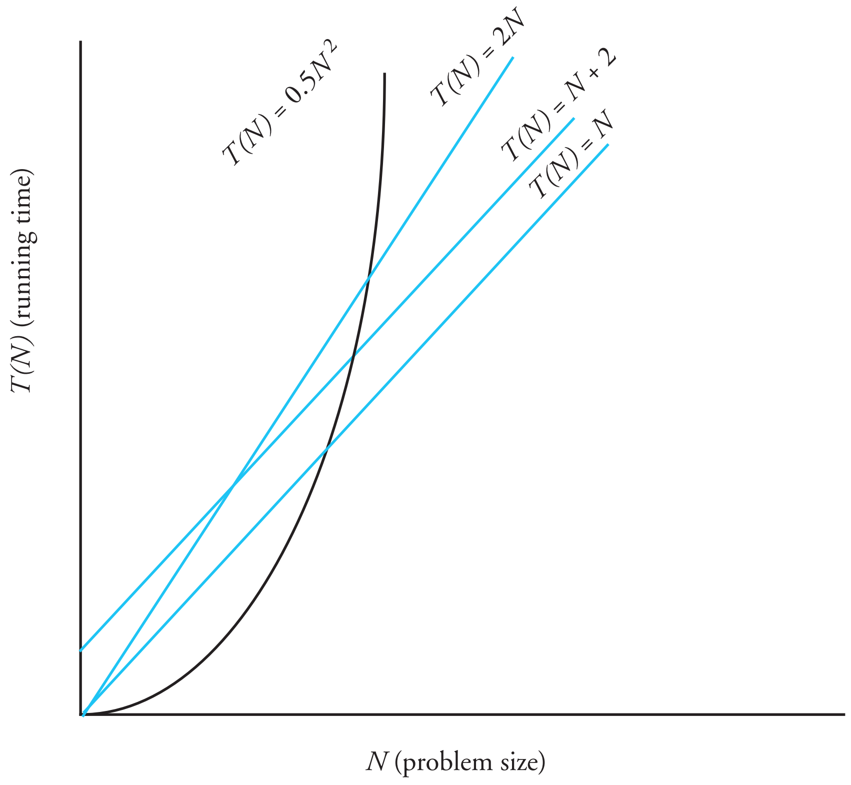
MyProgrammingLab™

Self-Test Exercises

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

19. 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 $\log_a N \leq c\,(\log_b N)$. Thus, there is no need to specify a base in $O(\log N)$. That is, $O(\log_a N)$ and $O(\log_b \text{N})$ mean the same thing.

Display 19.16 **Comparison of Running Times**



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. The generic `find` function can be used with any of the STL sequence container classes. Display 19.17 shows a sample use of the generic `find` function used with the class `vector<char>`. The function in Display 19.17 would behave exactly the same if we replaced `vector<char>` by `list<char>` throughout, or if we replaced `vector<char>` by any other sequence 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.

Display 19.17 **The Generic `find` Function** (part 1 of 2)

```
//Program to demonstrate use of the generic find function.
#include <iostream>
#include <vector>
#include <algorithm>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
using std::find;

int main( )
{
    vector<char> line;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != '\n';
    {
        line.push_back(next);
        cin.get(next);
    }

    vector<char>::const_iterator where;
    where = find(line.begin( ), line.end( ), 'e');
    //where is located at the first occurrence of 'e' in v.

    vector<char>::const_iterator p;
    cout << "You entered the following before you"
         << "entered your first line:\n";
    for (p = line.begin( ); p != where; p++)
        cout << *p;
    cout << endl;
```

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

(continued)

Display 19.17 **The Generic `find` Function** (part 2 of 2)

```
    cout << "You entered the following after that:\n";
    for (p = where; p != line.end( ); p++)
        cout << *p;
    cout << endl;

    cout << "End of demonstration.\n";
    return 0;
}
```

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( )`.

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 19.17. Sample Dialogue 2 in that display shows the situation when `find` does not find what it is looking for.

Does `find` work with absolutely any container? 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. Thus, 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. Because we have not discussed input and output iterators, however, we do not mention them in our function template declarations.

Remember 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.

Display 19.18 gives a sample of some nonmodifying generic functions in the STL. Display 19.18 uses a notation that is common when discussing container iterators. The iterator locations encountered in moving from an iterator `first` to—but not including—an iterator `last` are called the **range**. For example, the following `for` loop outputs all the elements in the range `[first,last)`:

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 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.

Notice that there are three search functions in Display 19.18: `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 function returns an iterator whereas 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 $O(\log N)$, whereas the `find` function does not require that the range be sorted, but 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`.

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 code 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;
```

Note that with the `binary_search` function, you are guaranteed that the implementation will use the binary search algorithm, which was discussed in Chapter 13. 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 13 and have not otherwise heard of a 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 searches that are relevant to the material in this chapter.

Display 19.18 Some Nonmodifying Generic Functions

These functions 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 that we have not covered in any detail.)

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

template <class ForwardIterator, class T>
int[4] count(ForwardIterator first, ForwardIterator last, const T& target);
//Traverse the range [first, last) and returns the number
//of elements equal to target.
//Time complexity: linear in the size of the range [first, last).

template <class ForwardIterator1, class ForwardIterator2>
bool equal(ForwardIterator1 first1, ForwardIterator1 last1,
                                  ForwardIterator2 first2);
//Returns true if [first1, last1) contains the same elements in the same
//order as the first last1-first1 elements starting at first2.
//Otherwise, returns false.
//Time complexity: linear in the size of the range [first, last).

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2);
//Checks to see if [first2, last2) is a subrange of [first1, last1).
//If so, it returns an iterator located in [first1, last1) at the start
//of the first match. Returns last1 if a match is not found.
//Time complexity: quadratic in the size of the range [first1, last1).

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

[4]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`.

MyProgrammingLab™

Self-Test Exercises

20. Replace all occurrences of the identifier `vector` with the identifier `list` in Display 19.17. Compile and run the program.

21. Suppose `v` is an object of the class `vector<int>`. Use the `search` generic function (Display 19.18) 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.)

Modifying Sequence Algorithms

Display 19.19 contains descriptions of some of the generic functions in the STL that change the contents of a container in some way.

Remember that adding or removing an element to or from a container 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 19.19 guarantee the values of some specific iterators; you can, of course, count on those guarantees no matter what the container is.

Display 19.19 **Some Modifying Generic Functions** (part 1 of 2)

```
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);
//last2 is such that 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);
```

(continued)

Display 19.19 **Some Modifying Generic Functions** (part 2 of 2)

```
//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).
```

MyProgrammingLab™

Self-Test Exercises

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

23. 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 19.20 shows a sample of the generic set operation functions defined in the STL. Note that generic algorithms assume that the containers store their elements in sorted order. The containers `set`, `map`, `multiset`, and `multimap` do store their elements in sorted order; therefore, all the functions in Display 19.20 apply to these four template class containers. Other containers, such as `vector`, do not store their elements in sorted order; 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 19.20 Set Operations

These functions work for `sets`*,* `maps`*,* `multisets`*, and* `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 to work, the elements in the container must be stored in sorted order. 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 that we have not covered in any detail.)*

```
template <class ForwardIterator1, class ForwardIterator2>
bool includes(ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2);
//Returns true if every element in the range [first2, last2) also occurs
//in the range [first1, last1). Otherwise, returns false.
//Time complexity: linear in the size of [first1, last1) plus [first2,
//last2).

template <class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator3>
void5 set_union(ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
                                       ForwardIterator3 result);
//Creates a sorted union of the two ranges [first1, last1) and [first2,
//last2). The union is stored starting at result.
//Time complexity: linear in the size of [first1, last1) plus [first2,
//last2).

template <class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator3>
void5 set_intersection(ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                                              ForwardIterator3 result);
//Creates a sorted intersection of the two ranges [first1, last1) and
//[first2, last2). The intersection is stored starting at result.
//Time complexity: linear in the size of [first1, last1) plus [first2,
//last2).

template <class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator3>
void5 set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
                                       ForwardIterator3 result);
//Creates a sorted set difference of the two ranges [first1, last1) and
//[first2, last2). The difference consists of the elements in the first
//range that are not in the second. The result is stored starting at
//result. Time complexity: linear in the size of [first1, last1) plus
//[first2, last2).
```

[5]Returns an iterator of type `ForwardIterator3` but can be used as a `void` function.

MyProgrammingLab™

Self-Test Exercise

24. 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 19.21 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 running time of $O(N \log N)$. Although it is beyond the scope of this book, it can be shown that you cannot write a sorting algorithm that is faster than $O(N \log N)$. So, this function guarantees that the sorting algorithm is as fast as possible, up to a constant multiple.

Display 19.21 Some Generic Sorting Algorithms

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
//Sorts the elements in the range [first, last) into ascending order.
//Time complexity: O(N log N), where N is the size of the range [first,
//last).

template <class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator3>
void merge(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                ForwardIterator3 result);
//Precondition: The ranges [first1, last1) and [first2, last2) are
//sorted.
//Action: Merges the two ranges into a sorted range [result, last3)
//where last3 = result + (last1 - first1) + (last2 - first2).
//Time complexity: linear in the size of the range [first1, last1)
//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 as follows.
  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 adapter 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.
```
using std::vector<int>::iterator;
  . . .
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 `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. To facilitate an efficient search for elements.
15. Yes, they can be of any type, although there is only one type for each object. The type parameter in the template class is the type of element 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. `mymap` will contain two entries. One is a map from `5` to `"c++"` and the other is a map from `4` to the default string, which is blank.
18. Just note that $aN + b \leq (a + b)N$, as long as $1 \leq N$.
19. 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 $\log_a N = c(\log_b N)$.
20. The programs should run exactly the same.
21.
```
#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";
```
22. No, you must have random-access iterators, and the `list` template class only has bidirectional iterators.
23. Yes, a random-access iterator is also a forward iterator.

24. 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.

Programming Projects

MyProgrammingLab™

Visit www.myprogramminglab.com to complete select exercises online and get instant feedback.

1. The point of this exercise is to demonstrate that an object that *behaves* like an iterator *is* an iterator. More precisely, if an object accesses some container and behaves like an iterator of a particular strength, then that object can be used as an iterator to manipulate the container with any of the generic functions that require an iterator having that strength. However, while the generic algorithms can be used with this container, the *member* functions, such as `begin` and `end`, will (of course) not be present (for example, in an array) unless they have been explicitly coded for that container. We will restrict this exercise to arrays of `double`, but the same message would hold true for any base type.
   a. Argue from the properties of random-access iterators that a pointer that points to an element of an array behaves exactly as a random-access iterator.
   b. Argue further that
      i) the name of the array is a pointer to double that points to the first element and so the array name can serve as a "begin" iterator and
      ii) (the array's name) + (the size of the array) can serve as an "end" pointer. (Of course, this points *one past* the end, as it should.)
   c. Write a short program in which you declare an array of `double` of size 10, and populate this array with 10 `double`s. Then call the `sort` generic algorithm with these pointer values as arguments and display the results.
2. This problem intends to illustrate removal of several instances of a particular item from a container with the `remove` generic function. A side effect is to examine the behavior of the generic function `remove` that comes with your compiler. (We have observed some variation in the behavior of `remove` from one compiler to another.) Before you start, look up the behavior of the `remove` generic algorithm as described by your favorite STL document or Web site. (For example, point your browser at `http://www.sgi.com/tech/stl/remove.html`. This worked as of the publication date.)
   a. Modify the array declaration in Programming Project 19.1 to include several elements with the same value (say, 4.0, but you can use any value for this exercise). Make sure that some of these are not all together. Use the modifying generic function `remove` (see Display 19.19) to remove all elements 4.0 (that is, the value you duplicated in building the array). Test.
   b. Use the array of `double` from part a to build a `list` and a `vector` that have the same contents as the array. Do this using the container constructor that takes two iterators as arguments. The `vector` and `list` classes each have a constructor that takes two iterators as arguments and initializes the `vector` or `list` to

the items in the iterator interval. The iterators can be located in any container, including in an array. Build the `vector` and `list` using the array name for the begin iterator and the name + array length as the end iterator for the two constructor arguments. Use the modifying algorithm `remove` (Display 19.19) to remove from the `list` and from the `vector` all elements equal to 4.0 (or the value you duplicated in building the array). Display the contents of the `vector` and `list` and explain the results.

c. Modify the code from part b to assign to an iterator variable of appropriate type the iterator value returned by the call to the `remove` generic function. Review the documentation for `remove` to be certain you know what these iterator values mean. Output the contents of the array, the `vector` and the `list`, `begin( )` to `end( )`, using the "begin" and "end" we described previously for the array. Output the contents of the two containers starting at the iterator returned from `remove` to the `end()` of each container. Explain your results.

3. 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 `x = y*z`. The Greek mathematician Erathosthenes (pronounced Er-ah-tos-thin-eeze) gave an algorithm for finding all prime numbers less than some integer *N*. This algorithm is called the *Sieve of Erathosthenes*. It 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, which is 3, is the next prime. The *multiples of 3 are not primes*. Cross these 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 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.

Improvements:

c. Actually, we do not need to go all the way to *N*. You 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.

4. 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 are not interested in who had the maximum and minimum grade, though, just the maximum, minimum, and average statistics.) Test your program.

5. Continuing Programming Project 19.4, 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 do not 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 is the run time for the `erase` and `insert` functions? Consider how the time efficiency of `erase` for a `list` affects the runtime 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.

6. a. Here is pseudocode for a program that inputs a value *n* from the user and then inserts *n* random numbers, ensuring that there are no duplicates:

```
Input n from user
Create vector v of type int
Loop i = 1 to n
   r = random integer between 0 and n-1
   Linearly search through v for value r
   if r is not in vector v then add r to the end of v
End Loop
Print out number of elements added to v
```

   Implement this program with your own linear search routine and add wrapper code that will time how long it takes to run. Test the program for different values of *n*. Depending on the speed of your system, you may need to input large values for *n* so that the program takes at least one second to run. Here is a sample that indicates how to calculate the difference in time: (`time.h` is a library that should be available on your version of C++).

```
#include <time.h>

time_t start,end;
double dif;

time (&start);                      // Record start time
// Rest of program goes here.
time (&end);                        // Record end time
dif = difftime(end,start);
cout << "It took " << dif << " seconds to execute. " << endl;
```

b. Next, create a second program that has the same behavior except that it uses an STL set to store the numbers instead of a vector:

```
Input n from user
Create set s of type int
Loop i = 1 to n
      r = random integer between 0 to n-1
      Use s.find(r) to search if r is already in the set
      if r is not in set s then add r to s
End Loop
Print out number of elements added to s
```

Time your new program with the same values of *n* that you used in the vector version. What do the results tell you about the Big-*O* run time of the `find( )` function for the set compared with linear search through the vector? Note that the `find( )` function is really redundant because insert has no effect if the element is already in the set. However, use the `find( )` function anyway to create a program comparable to the vector algorithm.

7. Modify your program from part a of Programming Project 19.6 so that the generic `find` function is used to search the vector for an existing value in place of your own code. You may wish to test your program with sample data to make sure that it is working correctly.

8. The field of information retrieval is concerned with finding relevant electronic documents based on a query. For example, given a group of keywords, a search engine retrieves Web pages (documents) and displays them in order, with the most relevant documents listed first. This technology requires a way to compare a document with the query to see which is most relevant to the query.

   A simple way to make this comparison is to compute the binary cosine coefficient. The coefficient is a value between 0 and 1, where 1 indicates that the query is very similar to the document and 0 indicates that the query has no keywords in common with the document. This approach treats each document as a set of words. For example, consider the following sample document:

   "Cows are big. Cows go moo. I love cows."

This document would be parsed into keywords where case is ignored and punctuation discarded and turned into the set containing the words "{cows, are, big, go, moo, i, love}". An identical process is performed on the query.

Once we have a query Q represented as a set of words and a document D represented as a set of words, the similarity between the query and document is computed by

$$Sim = \frac{|Q \cap D|}{\sqrt{|Q|}\sqrt{|D|}}$$

For example, if $D = $ {cows, are, big, go, moo, i, love} and $Q = $ {love, holstein, cows} then

$$Sim = \frac{|\{love, cows\}|}{\sqrt{|Q|}\sqrt{|D|}} = \frac{2}{\sqrt{3}\sqrt{7}} = 0.436$$

Write a program that allows the user to input a set of strings that represents a document and a set of strings that represents a query. (If you are more ambitious, you could write a program that parses an actual text file and computes the set of unique strings.) Represent the document and query as an STL set of strings. Then compute and print out the similarity between the query and document using the binary cosine coefficient. The `sqrt` function is in `cmath`. 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 <algorithm>
#include <set>
#include <string>
...
using std::insert_iterator;

set<string> A,B,C;
// Code below assumes strings have been inserted into A and B
// Note space between > > in line below
insert_iterator<set<string> > cIterator(C, C.begin( ));
set_intersection(A.begin( ), A.end( ),
                 B.begin( ),B.end( ),
                 cIterator);
// set C now contains the intersection of A and B
```

VideoNote **Solution to Programming Project 19.9**

9. Re-do or do for the first time Programming Project 5.8 in Chapter 5. This project asks you to approximate through simulation the probability that two or more people in the same room have the same birthday, for two to fifty people in the room.

However, instead of creating a solution that uses arrays, write a solution that uses a map. Over many trials (say, 5000), randomly assign birthdays (i.e., the numbers 1 through 365, assuming each number has an equal probability) to everyone in the room. Use a `map<int,int>` to map from the birthday (1–365) to a count of how many times that birthday occurs. Initially, each birthday should map to a count of 0. As the birthdays are randomly generated, increment the corresponding counter in the map. If a duplicate birthday is detected, then increment a counter for that trial. Over all trials this counter should indicate how many of those trials had a duplicate birthday. Divide the counter by the number of trials to get an estimated probability that two or more people share the same birthday for a given room size.

Your output should look the same as the output for Programming Project 5.8 in Chapter 5.

10. 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
Happy Feet
4
Happy Feet
5
Pirates of the Caribbean
3
Happy Feet
4
Pirates of the Caribbean
4
Flags of Our Fathers
5
Gigli
1
```

Write a program that reads in 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:

```
Happy Feet: 3 reviews, average of 4.3 / 5
Pirates of the Caribbean: 2 reviews, average of 3.5 / 5
Flags of Our Father: 1 review, average of 5 / 5
Gigli: 1 review, average of 1 / 5
```

Use a map or multiple maps to generate the output. Your map 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.

11. Write a program that outputs a histogram of grades for an assignment given to a class of students. The program should input each student's grade as an integer and store the grade in a vector. Grades should be entered until the user enters −1 for a grade. Use a map from an `int` to an `int` to compute the histogram. The first integer in the map represents the grade, and the second integer represents the number of times that grade occurred. Output the histogram to the console. See Programming Project 5.7 for information on how to compute a histogram. There should be no restrictions on the minimum and maximum grade for this programming project.

VideoNote
Solution to Programming Project 19.12

12. Consider a text file of names, with one name per line, that has been compiled from several different sources. A sample is shown in the following:

```
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 do not 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.

13. Reverse Polish Notation (RPN) or postfix notation is a format to specify mathematical expressions. In RPN the operator comes after the operands instead of the more common format in which the operator is between the operands (this is called infix notation). Starting with an empty stack, a RPN calculator can be implemented with the following rules:

- If a number is input, push it on the stack.
- If `+` is input, then pop the last two operands off the stack, add them, and push the result on the stack.
- If `-` is input, then pop `value1`, pop `value2`, then push `value2 - value1` on the stack.
- If `*` is input, then pop the last two operands off the stack, multiply them, and push the result on the stack.
- If `/` is input, then pop `value1`, pop `value2`, then push `value2 / value1` on the stack.
- If `q` is input, then stop inputting values, print out the top of the stack, and exit the program.

Use the `stack` template class to implement a RPN calculator. Output an appropriate error message if there are not two operands on the stack when given an operator. Here is sample input and output that is equivalent to $((10 - (2 + 3))*2)/5$:

```
10
2
3
+
-
2
*
5
/
q

The top of the stack is: 2
```