6

Abstract Container Types

> This chapter provides both an extension of and completion to Chapters 3 and 4. We continue the discussion of types started in Chapter 3 by presenting more information on the string and vector types as well as presenting other container types provided by the C++ standard library. In addition, we continue the discussion of operators and expressions started in Chapter 4 by presenting the operations supported for objects of the container types.

A sequence container holds an ordered collection of elements of a single type. The two primary sequence containers are the vector and list types. (A third sequence container, deque — pronounced *deck* — provides the same behavior as a vector but is specialized to support efficient insertion and deletion of its first element. A deque is preferred over a vector, for example, in the implementation of a queue, an abstraction in which the first element is retrieved each time. In the remainder of the text, whenever we describe the operations supported by a vector, those operations are supported by a deque as well.)

An associative container supports efficient query as to the presence and retrieval of an element. The two primary associative container types are the map and set. A map is a key/value pair: the key is used for look-up, the value contains the data we wish to make use of. A telephone directory, for example, is well-supported by a map: the key is the individual's name, the value is the associated phone number.

A set contains a single key value, and supports the efficient query of whether it is present or not. For example, a text query system might build a set of words to exclude, such as *the*, *and*, *but*, and so on, when building up a database of the words present within a text. The program would read each word of the text in turn, check if it is in the set of excluded words, and either discard or enter the word within the database depending on the result of the query.

Both the map and set can contain only a single occurrence of each key. A multimap and multiset support multiple occurrences of the same key. Our telephone directory,

for example, is likely to need to support multiple listings for a single individual. One method of implementing that is with the use of a multimap.

In the sections that follow, we look at the container types in detail, motivating the discussion through a progressive implementation of a small text query program.

6.1 Our Text Query System

What does a text query system consist of?

1. An arbitrary text file indicated by the user.
2. A boolean query facility in which the user can search for a word or sequence of adjacent words within the text.

If the word or sequence of adjacent words are found, the number of occurrences of each word and word sequence is displayed. If the user wishes, the sentence(s) within which the word or word sequence occurs is also displayed. For example, if the user wished to find all references to either the Civil War or Civil Rights, the query might look as follows[1]:

```
Civil && ( War || Rights )
```

The result of the query might look as follows:

```
Civil:12 occurrences
War: 48 occurrences
Rights:1 occurrence

Civil && War:1 occurrence
Civil && Rights:1 occurrence

(8) Civility, of course, is not to be confused with
Civil Rights, nor should it lead to Civil War.
```

[1]Note: to simplify our implementation, we require a space separating each word, including parentheses and the boolean operators. So that

```
(War || Rights)
```

will not be understood nor

```
Civil&&(War||Rights)
```

While this is unreasonable in a real-world system in which the user's convenience always overrides convenience for the implementors, we believe it is more than acceptable in a primarily introductory text such as this.

where (8) represents the sentence number of the text. Our system must be smart enough not to display the same sentence multiple times. Moreover, multiple sentences should be displayed in ascending order (that is, sentence 7 should always be displayed prior to sentence 9).

What are the tasks that our program needs to support?

1. It must allow the user to indicate the name of a text file to open, then open and read the text.
2. It must internally organize the text file such that it can identify the number of occurrences of each word in terms of the sentence it occurs in and its position within that sentence.
3. It must support some form of boolean query language. In our case, it will support the following:

   `&&` both words are not only present but adjacent within a sentence.

   `||` one or both words are present in a sentence.

   `!` the word is not present in a sentence.

   `()` a means of subgrouping a query.

Thus, one can write

```
Lincoln
```

to find all the sentences in which the word Lincoln occurs, or

```
! Lincoln
```

to find all the sentences in which the word Lincoln does not occur, or

```
( Abe || Abraham ) && Lincoln
```

to limit the sentences selected to those explicitly referring to Abe Lincoln or Abraham Lincoln.

We provide two implementations to our system. In this chapter, we provide an implementation solving the problem of retrieving and storing the text file as a map of word entries and their associated line and column locations. To exercise this solution, we provide a single word query system. In Chapter 17, we provide an implementation of the full query system supporting the relational operators such as we discussed in the paragraphs above. We defer its implementation until then because the solution involves the use of an object-oriented Query class hierarchy.

For exposition within the text, we use the following six lines from an unpublished children's story Stan has written[2]:

```
Alice Emma has long flowing red hair. Her Daddy says
```

```
When the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells
her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

The internal storage of the above text to support individual queries, at the end of our processing, looks as follows (this involves reading in the individual lines of text, separating them into the individual words, stripping out punctuation, eliminating capitalization, providing some minimal support of suffixing, and eliminating semantically neutral words such as *and, a, the*):

```
alice ((0,0))
alive ((1,10))
almost ((1,9))
ask ((5,2))
beautiful ((2,7))
bird ((2,3),(2,9))
blow ((1,3))
daddy ((0,8),(3,3),(5,5))
emma ((0,1))
fiery ((2,2),(2,8))
flight ((2,5))
flowing ((0,4))
hair ((0,6),(1,6))
has ((0,2))
like ((2,0))
long ((0,3))
look ((1,8))
magical ((3,0))
mean ((5,4))
more ((4,12))
red ((0,5))
same ((4,5))
say ((0,9))
she ((4,0),(5,1))
shush ((3,4))
shyly ((5,0))
```

[2] We have provided the full text of Herman Melville's wonderful story, Bartleby the Scrivner, at our ftp site, together with the full text of both versions of the program.

```
such ((3,8))
tell ((2,11),(4,1),(4,10))
there ((3,5),(5,7))
thing ((3,9))
through ((1,4))
time ((4,6))
untamed ((3,2))
wanting ((4,7))
wind ((1,2))
```

The following is a sample query session using the program implemented within this chapter (user entries are in italics):

```
please enter file name: alice_emma

warning! unable to open word exclusion file! -- using default
set

enter a word against which to search the text.
to quit, enter a single character ==>  alice

alice occurs 1 time:

        ( line 1 ) Alice Emma has long flowing red hair. Her
Daddy says

enter a word against which to search the text.
to quit, enter a single character ==>  daddy

daddy occurs 3 times:

        ( line 1 ) Alice Emma has long flowing red hair. Her
Daddy says
        ( line 4 ) magical but untamed. "Daddy, shush, there
is no such thing,"
       ( line 6 ) Shyly, she asks, "I mean, Daddy, is there?"

enter a word against which to search the text.
to quit, enter a single character ==>  phoenix

Sorry. There are no entries for phoenix.
```

```
enter a word against which to search the text.
to quit, enter a single character ==>  .
Ok, bye!
```

In order to easily implement this program, we need to look in detail at the standard library container types, as well as revisit the string class introduced in Chapter 3.

6.2 A vector or a list?

One of the first things our program must do is store an unknown number of words from a text file. The words will be stored in turn as string objects. Our first question is, should we store the words in a sequence or associative container?

At some point we will need to support the query as to the presence of a word and, if present, to retrieve its associated occurrences within the text. Because we are both searching for and retrieving a value, an associative map is the most appropriate container type to support this.

However, prior to that, we need to simply store the input text for processing — that is, to strip out punctuation, deal with suffixing, and so on. For this preliminary pass, a sequence not an associative container is required. The question is, should it be a vector or a list?

If you have programmed in C, or in prestandard C++, your rule of thumb in choosing is probably something as follows: if the number of elements to be stored is known at compile-time, use an array. If the number of elements to be stored is unknown or likely to vary widely, then use a list, dynamically allocating storage for each object and attaching that object to the list in turn.

This rule of thumb, however, does not hold for the sequence container types: the vector, deque, and list all grow dynamically. The criteria for choosing among these three is primarily concerned with the insertion characteristics and subsequent access requirements of the elements.

A vector represents a contiguous area of memory in which each element is stored in turn. Random access into a vector — this is, accessing element 5, then 15, then 7, and so on — is very efficient since each access is a fixed offset from the beginning of the vector. Insertion of an element at any position other than the back of the vector, however, is inefficient since it requires each element to the right of the inserted element to be copied. Similarly, a deletion of any element other than the last element of the vector is inefficient because each element to the right of the deleted element has to be copied. This can be particularly expensive for large, complex class objects. (A deque also represents a contiguous area of memory; however, unlike a vector, it supports the efficient insertion and deletion of elements at its front as well. It achieves this through a two-level array structure in which one level represents the actual container and a second

level addresses the front and back of the container.)

A list represents non-contiguous memory doubly-linked through a pair of pointers addressing the elements to the front and back allowing for both forward and backward traversal. Insertion and deletion of elements at any position within the list is efficient: the pointers need to be reassigned, but no elements need to be moved by copying. Random access, on the other hand, is not well supported: accessing an element requires traversal of intervening elements. In addition, there is the space overhead of the two pointers per element.

Here are some criteria for choosing a sequence container type:

- if we require random access into a container, a vector is the clear choice over a list;
- if we know the number of elements we need to store, a vector is again to be preferred over a list;
- if we need to insert and delete elements other than at the two ends of the container, a list is the clear choice over a vector;
- unless we need to insert or delete elements at the front of the container, a vector is preferable over a deque.

What if we need both to randomly access and randomly insert and delete elements? The trade-off is between the cost of the random access versus that of copying contiguous elements to the right or left. In general, the predominant operation of the application (the search or insertion) should determine the choice of container type. (This may require profiling the performance of both container types.) If neither performance is satisfactory, it may be necessary to design a more complex data structure of our own.

How do we decide which to choose when we do not know the number of elements we need to store (that is, the container is going to grow dynamically), and there is no need either for random access or insertion other than at the back? Is a list or vector in this case significantly more efficient? (We'll need to postpone an answer to this until the next section.)

A list grows in a straightforward manner: each time a new object is inserted into the list, the front pointer and back pointer of the two elements between which the new object is being inserted are reassigned to point to the new object. The front and back pointer of the new object, in turn, are initialized to address these two elements. The list holds only the storage necessary for the elements it contains. The overhead is two-fold: the two additional pointers associated with each value, and the indirect access of the value through a pointer.

The representation and overhead of a dynamic vector is more complex. We look at that in the next section.

Exercise 6.1

Which is the more appropriate, a vector, deque, or a list, for the following program tasks, or is neither preferred?

```
(a) Read in an unknown number of words from a file for the
purpose
    of generating random English language sentences.

(b) Read in a fixed number of words, inserting them in the
container
    alphabetically as they are entered.

(c) Read in an unknown number of words. Always insert new
words at the
    back. Remove the next value from the front.

(d) Read in an unknown number of integers from a file. Sort
the numbers,
    then print them to standard output.
```

6.3 How a vector Grows Itself

For a vector to dynamically grow, it must (a) allocate the memory to hold the new sequence, (b) copy the elements of the old sequence in turn, and (c) deallocate the old memory. Moreover, if the elements are class objects, the copy and deallocation may require the invocation of the class copy constructor and destructor on each element in turn. Since a list simply links in the new elements each time the container grows, there seems little question that a list is the more efficient of the two container types in its support of dynamic growth. But in practice this is not the case. Let's see why.

In order to be of even minimum efficiency, the vector cannot actually regrow itself with each individual insertion. Rather, when the vector needs to grow itself, it allocates additional storage capacity beyond its immediate need — it holds this in reserve. (The exact amount of additional capacity allocated is implementation defined.) This allows for a significantly more efficient regrowing of the container — so much so, in fact, that for small objects, a vector in practice turns out to grow more efficiently than a list. Let's look at some examples under the Rogue Wave implementation of the C++ standard library. But first, let's make clear the distinction between the capacity and size of a con-

tainer.

Capacity is the total number of elements that can be added to a container before it needs to regrow itself. (Capacity is only associated with a container in which storage is contiguous: for example, a vector, deque, or string. A list does not require capacity.) To discover the capacity of a vector, we invoke its `capacity()` operation. Size, on the other hand, is the number of elements a container currently holds. To retrieve the current size of a container, we invoke its `size()` operation. For example,

```
#include <vector>
#include <iostream>

int main()
{
vector< int > ivec;
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;

for ( int ix = 0; ix < 24; ++ix ) {
      ivec.push_back( ix );
      cout << "ivec: size: " << ivec.size()
           << " capacity: "  << ivec.capacity() << endl;
}
}
```

Under the Rogue Wave implementation, both the size and capacity of `ivec` after its definition is 0. On inserting the first element, however, `ivec`'s capacity is 256 while its size is 1. This means that 256 elements can be added to `ivec` before it needs to regrow itself. When we do insert a 256th element, the vector regrows itself in the following way: (1) it allocates double its current capacity, (2) copies its current values into the new allocated memory, and (3) deallocates its previous memory. As we'll see in a moment, the larger and more complex the data type, the less efficient the vector becomes as compared to a list. Here is a table of data types, their sizes, and the initial capacity of their associated vector:

data type	size in bytes	capacity after initial insertion
int	4	256
double	8	128
simple class #1	12	85
string	12	85
large simple class	8000	1
large complex class	8000	1

As you can see, under the Rogue Wave implementation, a default capacity of elements near to or equal 1024 bytes is allocated with a first insertion, then doubled with each reallocation. For a large data type, with a small capacity, the reallocation and copying of the elements becomes the primary overhead in the use of the vector. (When we speak of a complex class, we mean a class that provides both a copy constructor and copy assignment operator.) Here are the times, in seconds, for inserting 10 million elements of the above types in both a list and vector:

Time in seconds to insert 10 million elements

data type	list	vector
int	10.38s	3.76s
double	10.72s	3.95s
simple class	12.31s	5.89s
string	14.42s	11.80s

Time in seconds to insert 10 thousand elements*

data type	list	vector
large simple class	0.36s	2.23s
large complex class	2.37s	6.70s

```
* only 10 thousand elements due to slowness
```

As you can see, for small data types, a vector performs considerably better than a list, whereas for large data objects, the reverse is true: a list performs considerably better. This is due to the need to regrow and copy the elements of the vector. The size of the data type, however, is not the only criteria affecting the performance of the container. In addition, the complexity of the data type also affects the performance of element insertion. Why?

The insertion of an element, either for a list or vector, requires invocation of the copy constructor for a class that defines one (a copy constructor, recall, initializes one class object with another object of its type — see Section 2.2 for an initial discussion, and Section 14.5 for a detailed discussion). This explains the difference in cost between the simple class and string class list insertion. The simple class objects and large simple class objects are inserted through a bitwise copy (that is, the bits of the object are copied into the bits of the second object) while the string class objects and large complex class objects are inserted through an invocation of the string copy constructor.

In addition, however, the vector must invoke the copy constructor for each element with each reallocation of its memory. Moreover, the deallocation of the previous memory requires the invocation of the associated class destructor on each element (again, see Section 2.2 for an initial discussion of a destructor). The more frequently the vector is required to regrow itself, the costlier the element insertion becomes.

One solution, of course, is to switch from a vector to a list when the cost of the vector becomes prohibitive. An alternative often preferable solution is to store large or

complex class objects indirectly by pointer. For example, when we store the complex class object by pointer, the cost of inserting 10 thousand elements within the vector goes down from 6.70s to 0.82s. Why? The capacity increases from 1 to 256, so the number of reallocations drops considerably. Secondly, the copy and deallocation of a pointer to class object does not require the invocation of either the copy constructor or destructor of the class.

The `reserve()` operation allows the programmer to set the container's capacity to an explicit value. For example,

```
int main {
   vector< string > svec;
   svec.reserve( 32 ); // sets capacity to 32
   // ...
}
```

results in `svec` having a size of zero elements, but a capacity of 32. What we found by experimentation, however, is that adjusting the capacity of a vector with a default capacity other than 1 seemed to always cause the performance to degrade: for example, both with the string and double vectors, increasing the capacity through `reserve()` resulted in a worse performance. On the other hand, increasing the capacity of a large, complex class provided significant performance improvement:

Time in seconds to insert 10 thousand elements with adjustments to capacity

(Non-simple class: 8000 bytes with both a copy constructor and destructor)

capacity	time in seconds
default of 1	6.70s
4,096	5.55s
8,192	4.44s
10,000	2.22s

For our text query system, then, we'll use a vector to contain our string objects with its default associated capacity. Although the vector grows dynamically as we insert an unknown number of strings within it, as our timings shows, it still performs slightly better than a list. Before we get to our actual implementation, let's review how we can define a container object.

Exercise 6.2

Explain the difference between a vector's capacity and its size. Why is it necessary to

support the notion of capacity in a container storing elements contiguously, but not, for example, in a list?

Exercise 6.3

Why is it more efficient to store a collection of a large, complex class object by pointer, but less efficient to store a collection of integer objects by pointer?

Exercise 6.4

In the following situations, which is the more appropriate container type, a list or a vector? In each case, an unknown number of elements are inserted. Explain why

```
(a) integer values
(b) pointers to a large, complex class object
(c) large, complex class objects
```

6.4 Defining a Sequence Container

To define a container object, we must first include its associated header file, which is one of

```
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
```

The definition of a container object begins with the name of the container type followed by the actual type of the elements to be contained.[3] For example,

```
vector< string > svec;
list< int > ilist;
```

[3] Implementations that do not currently support default template parameters require a second argument specifying the *allocator*. Under these implementations, the above two definitions are declared as follows:

```
vector< string, allocator > svec;
list< int, allocator > ilist;
```

The allocator class encapsulates the abstraction of allocating and deleting dynamic memory. It is predefined by the standard library and uses the `new` and `delete` operators. The use of an allocation class serves two purposes: by shielding the containers from the details of this or that memory allocation strategy, it (1) simplifies the implementation of the container, and (2) makes it possible for the programmer to implement and/or specify alternative memory allocation strategies, such as the use of shared memory.

defines svec to be an empty vector of string objects and ilist to be an empty list of objects of type int. Both svec and ilist are empty. To confirm that, we can invoke the empty() operator. For example,

```
if ( svec.empty() != true )
     ; // oops, something wrong
```

The simplest method of element insertion is push_back(), which inserts the element to the back of the container. For example,

```
string text_word;
while ( cin >> text_word )
        svec.push_back( text_word );
```

reads one string at a time into text_word from standard input. push_back() then inserts a copy of the text_word string into svec. The list container also supports push_front(), which inserts the new element at the front of the list. For example, given the following built-in array of type int:

```
int ia[ 4 ] = { 0, 1, 2, 3 };
```

use of push_back()

```
for ( int ix = 0; ix < 4; ++ix )
      ilist.push_back( ia[ ix ] );
```

creates the sequence 0,1,2,3 while the use of push_front()

```
for ( int ix = 0; ix < 4; ++ix )
      ilist.push_front( ia[ ix ] );
```

creates the sequence 3,2,1,0 within ilist[4].

Alternatively, we may wish to specify an explicit size for the container. The size can be either a constant or nonconstant expression:

```
#include <list>
#include <vector>
#include <string>

extern int get_word_count( string file_name );
const int list_size = 64;
```

[4] If the use of push_front() becomes the predominant container activity, a deque performs significantly more efficiently than a vector, and should be preferred.

```
list< int > ilist( list_size );
vector< string > svec( get_word_count( string("War and
Peace")));
```

Each element within the container is initialized with the associated default value for its type. For an integer, a default value of 0 is used to initialize each element. For the string class, each element is initialized with the associated string default constructor.

Rather than initializing each element to its associated default value, we can specify a value with which to initialize each element. For example,

```
list< int > ilist( list_size, -1 );
vector< string > svec( 24, "pooh" );
```

In addition to providing an initial size, we can physically resize the container through the `resize()` operation. For example, when we write

```
svec.resize( 2 * svec.size() );
```

we double the current size of `svec`. Each new element is initialized with the default value associated with the underlying type of the element. If we wish to initialize each new element to some other value, we can specify that value as a second argument:

```
// initialize each new element to "piglet"
svec.resize( 2 * svec.size(), "piglet" );
```

By the way, what is the capacity of the original definition of `svec`? It has an initial size of 24 elements. What is its likely initial capacity? That's right, `svec` has a capacity of 24 as well. In general, the minimal capacity of a vector is its current size. When we double the size of a vector, in general we double its capacity as well.

We can also initialize a new container object with a copy of an already existing container object. For example,

```
vector< string > svec2( svec );
list< int > ilist2( ilist );
```

Each container supports a set of relational operators against which two containers can be compared: equality, inequality, less-than, greater-than, less-than-or-equal, and greater-than-or-equal). The comparison is based on a pairwise comparison of the elements of the two containers. If all the elements are equal and both contain the same number of elements, the two containers are equal; otherwise, they are unequal. A comparison of the first non-equal element determines the less-than or greater-than relationship of the two containers. For example, here is the output of a program comparing 5 vectors:

```
ivec1: 1 3 5 7 9 12
ivec2: 0 1 1 2 3 5 8 13
ivec3: 1 3 9
ivec4: 1 3 5 7
ivec5: 2 4

// first unequal element: 1, 0
// ivec1 greater than ivec2
ivec1 < ivec2 false
ivec2 < ivec1 true

// first unequal element 5, 9
ivec1 < ivec3 true

// all elements equal but ivec4 has less elements
// so ivec4 is less than ivec1
ivec1 < ivec4 false

// first unequal element: 1, 2
ivec1 < ivec5 true

ivec1 == ivec1 true
ivec1 == ivec4 false
ivec1 != ivec4 true

ivec1 > ivec2 true
ivec3 > ivec1 true
ivec5 > ivec2 true
```

There are three constraints as to the types of containers that we may define (in practice, these only pertain to user-defined class types):

- the element type must support the equality operator;
- the element type must support the less-than operator (all the relational operators discussed above are implemented using these two operators);
- the element type must support a default value (again, for a class type, this is spoken of as a default constructor);

The predefined data types, including pointers, all meet these constraints, as do all the class types provided by the C++ standard library.

Exercise 6.5

Explain what the following program does:

```
#include <string>
#include <vector>
#include <iostream>

int main()
{
     vector<string> svec;
     svec.reserve( 1024 );

     string text_word;
     while ( cin >> text_word )
             svec.push_back( text_word );

     svec.resize( svec.size()+svec.size()/2 );
     // ...
}
```

Exercise 6.6

Can a container have a capacity less than its size? Is a capacity equal to its size desirable? Initially? After an element is inserted? Why or why not?

Exercise 6.7

In Exercise 6.5, if the program reads in 256 words, what is its likely capacity after it is resized? If it reads in 512? 1000? 1048?

Exercise 6.8

Given the following class definitions, which are not able to be used for defining a vector?

```
(a) class cl1 {      (b) class cl2 {
    public:                public:

        cl1( int=0 ); cl2( int=0 );
        bool operator==();bool operator!=();
        bool operator!=(); bool operator<=();
        bool operator<=();// ...
        bool operator<();};
```

```
// ...
};

(c) class cl3 {
    public:
        int ival;
    };

tor==();
```

```
(d) class cl4 {
    public:
        cl4( int, int=0 );
        bool operator==();
             bool opera-

             // ...
        };
```

6.5 Iterators

An iterator provides a general method of successively accessing each element within any of the sequential or associative container types. For example, let `iter` be an iterator into any container type, then

```
++iter;
```

advances the iterator to address the next element of the container, and

```
*iter;
```

returns the value of the element addressed by the iterator.

Each container type provides both a `begin()` and `end()` member function.

- `begin()` returns an iterator addressing the first element of the container.
- `end()` returns an iterator addressing one past the last element of the container.

To iterate over the elements of any container type, we write:

```
for ( iter = container.begin();
      iter != container.end();
      ++iter )
{
    do_something_with_element( *iter );
}
```

The declaration of an iterator can look somewhat intimidating because of the template and nested class syntax. For example, here is the definition of a pair of iterators to a vector of string elements:

```
// vector<string> vec;
vector<string>::iterator iter = vec.begin();
vector<string>::iterator iter_end = vec.end();
```

`iterator` is a typedef defined within the vector class. The syntax

```
vector<string>::iterator
```

references the `iterator` typedef nested in the vector class holding elements of type string.

To print each string element to standard out, we write:

```
for( ; iter != iter_end; ++iter )
     cout << *iter << "\n";
```

where, of course, `*iter` evaluates to the actual string object.

In addition to the *iterator* type, each container also defines a *const_iterator* type. The latter is necessary in order to traverse a `const` container. A const_iterator permits read-only access of the underlying elements of the container. For example,

```
#include <vector >
void even_odd( const vector<int> *pvec,
 vector<int> *pvec_even,
 vector<int> *pvec_odd )
{
// must declare a const_iterator to traverse pvec
vector<int>::const_iterator c_iter = pvec->begin();
vector<int>::const_iterator c_iter_end = pvec->end();

for ( ; c_iter != c_iter_end; ++c_iter )
 if ( *c_iter % 2 )
      pvec_even->push_back( *c_iter );
 else pvec_odd->push_back( *c_iter );
}
```

Finally, what if we wish to look at some subset of the elements, perhaps every other element, or every third element, or to begin stepping through the elements starting from the middle? We can offset from an iterator's current position using scalar arithmetic. For example,

```
vector<int>::iterator iter = vec->begin()+vec.size()/2;
```

sets `iter` to address the middle element of `vec`, while

```
iter += 2;
```

advances `iter` two elements.

Iterator arithmetic works only with a vector and deque, not a list since the list elements are not stored contiguously in memory. For example,

```
ilist.begin() + 2;
```

is not correct because advancing two elements in a list requires following the internal *next* pointer twice. With a vector or deque, advancing two elements requires adding the size of two elements to the current address value (Section 3.3 provides a discussion of pointer arithmetic).

A container object can also be initialized with a pair of iterators marking off the first and one past the last element to be copied. For example, given

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
     vector<string> svec;
     string intext;

     while ( cin >> intext )
             svec.push_back( intext );

     // process svec ...
}
```

we can define a new vector to copy all or a subset of the elements of `svec`:

```
int main()
{
vector<string> svec;
// ...

// initialize svec2 with all of svec
vector<string> svec2( svec.begin(), svec.end() );

// initialize svec3 with first half of svec
vector<string>::iterator it =
  svec.begin() + svec.size()/2;
vector<string> svec3( svec.begin, it );
```

```
// process vectors ...

}
```

Using the special `istream_iterator` type (discussed in detail in Chapter 20 on the iostream library), we can more directly insert the text elements into `svec`:

```
#include <vector>
#include <string>
#include < iterator >

int main()
{
// input stream iterator tied to standard input
istream_iterator<string> infile( cin );

// input stream iterator marking end-of-stream
istream_iterator<string> eos;

// initialize svec with values entered through cin;
vector<string> svec( infile, eos );

// process svec
}
```

In addition to a pair of iterators, two pointers into a built-in array can also be used as element range markers. For example, given the following array of string objects,

```
#include <string>
string words[4] = {
  "stately", "plump", "buck", "mulligan"
};
```

we can initialize a vector of string by passing a pointer to the first element of the `words` array, and a second pointer *one past* the last string element:

```
vector<string> vwords( words, words+4 );
```

The second pointer serves as a stopping condition; the object it addresses (usually one past the last object within a container or array) is not included in the elements to be copied or traversed.

Similarly, we can initialize a list of `int` elements as follows:

```
int ia[6] = { 0, 1, 2, 3, 4, 5 };
list< int > ilist( ia, ia+6 );
```

In Chapter 12, in our discussion of the generic algorithms, we revisit iterators in a bit more detail. For now, we've introduced them sufficiently to use them in our text query system implementation. But before we turn to that, we need to review some additional operations supported by the container types.

Exercise 6.9

Which, if any, of the following iterator uses are in error?

```
const vector< int > ivec;
vector< string >    svec;
list< int >         ilist;

(a) vector<int>::iterator it = ivec.begin();
(b) list<int>::iterator it = ilist.begin()+2;
(c) vector<string>::iterator it = &svec[0];
(d) for ( vector<string>::iterator it = svec.begin();
          it != 0; ++it )
             // ...
```

Exercise 6.10

Which, if any, of the following iterator uses are in error?

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville", "Vicksburg",
    "Meridian", "Chancellorsville" };

(a) vector<string> svec( sa, &sa[6] );
(b) list<int> ilist( ia+4, ia+6 );
(c) list<int> ilist2( ilist.begin(), ilist.begin()+2 );
(d) vector<int> ivec( &ia[0], ia+8 );
(e) list<string> slist( sa+6, sa );
(f) vector<string> svec2( sa, sa+6 );
```

6.6 Sequence Container Operations

The `push_back()` method provides a convenient shorthand notation for inserting a single element at the end of a sequence container. But what if we wish to insert an element at some other position within the container? Or if we wish to insert a sequence of elements, either at the end or at some other position within the container? In these cases, we would use the more general set of insertion methods.

For example, to insert an element at the beginning of a container, we would do the following:

```
vector< string > svec;
list< string > slist;
string spouse( "Beth" );

slist.insert( slist.begin(), spouse );
svec.insert(  svec.begin(), spouse  );
```

where the first argument to `insert()` is the position (an iterator addressing some position within the container) and the second argument to `insert()` is the value to be inserted. The value is inserted in front of the position addressed by the iterator. A more random insertion might be programmed as follows:

```
string son( "Danny" );

list<string>::iterator iter;
iter = find( slist.begin(), slist.end(), son );

slist.insert( iter, spouse );
```

where `find()` either returns the position within the container at which the element is found, or else returns the `end()` iterator of the container to indicate the search failed. (We'll come back to `find()` at the end of the next section.) As you might have guessed, the `push_back()` method is a shorthand notation for the following call:

```
// equivalent to: slist.push_back( value );
slist.insert( slist.end(), value );
```

A second form of the `insert()` method supports inserting a specified count of elements at some position. For example, if we wished to insert ten Annas at the beginning of a vector, we would do the following:

```
vector<string> svec;
...
```

```
string anna( "Anna" );
svec.insert( svec.begin(), 10, anna );
```

The final form of the `insert()` method supports inserting a range of elements into the container. For example, given the following array of strings:

```
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
```

we can insert all or a subset of the array elements into our string vector:

```
svec.insert( svec.begin(), sarray, sarray+4 );
svec.insert( svec.begin() + svec.size()/2,
             sarray+2, sarray+4);
```

Alternatively, we can mark a range of values through a pair of iterators, either of another vector of string elements:

```
svec_two.insert( svec_two.begin() + svec_two.size()/2,
                 svec.begin(), svec.end() );
```

or, more generally, any container of string objects: [5]

```
list< string > slist;
// ...
list< string >::iterator iter =
  find( slist.begin(), slist.end(), stringVal );
slist.insert( iter, svec.begin(), svec.end() );
```

6.6.1 Deletion

The general form of element deletion within a container is a pair of `erase()` methods, one to delete a single element, the second to delete a range of elements marked off by a pair of iterators. A shorthand method of deleting the last element of a container is supported by the `pop_back()` method.

For example, to delete a specific element within the container, you simply invoke `erase()` with an iterator indicating its position. In the following code fragment, we use the generic `find()` algorithm to retrieve the iterator to the element we wish to delete, and, if the element is present within the list, pass its position to `erase()`.

```
string searchValue( "Quasimodo" );
list< string >::iterator iter =
```

[5] This last form requires that your compiler support template member functions. If your compiler does not as yet support this feature of Standard C++, then the two container objects must be of the same type, such as two vectors or two lists holding the same element type.

```
find( slist.begin(), slist.end(), searchValue );
if ( iter != slist.end() )
  slist.erase( iter );
```

To delete all the elements in a container, or a subset marked off by a pair of iterators, we can do the following:

```
// delete all the elements within the container
slist.erase( slist.begin(), slist.end() );

// delete the range of elements marked off by iterators
list< string >::iterator first, last;

first = find( slist.begin(), slist.end(), val1 );
last  = find(slist.begin(), slist.end(), val2 );

// ... check validity of first and last

slist.erase( first, last );
```

Finally, complementing the `push_back()` method that inserts an element at the end of a container, the `pop_back()` method deletes the end element of a container. For example,

```
vector< string >::iterator iter = buffer.begin();
for ( ; iter != buffer.end(), iter++ )
{
slist.push_back( *iter );
if ( ! do_something( slist ))
 slist.pop_back();
}
```

6.6.2 Assignment and Swap

What happens when we assign one container to another? The assignment operator copies the elements of the right-hand container object into the left-hand container in turn using the assignment operator of the element type of the container. What if the two containers are of unequal size? For example,

```
// svec1 contains 10 elements
// svec2 contains 24 elements
// after assignment, both hold 24 elements
svec1 != svec2;
```

The target of the assignment, `svec1` in our example, now holds the same number of elements as container from which the elements are copied (`svec2` in our example). The previous ten elements contained within `svec1` are erased. (In the case of `svec1`, the string destructor is applied to each of the ten string elements.)

`swap()` can be thought of as the complement to the assignment operator. When we write

```
slist1.swap( slist2 );
```

`slist1` now contains 24 string elements that were copied using the string assignment operator the same as if we had written

```
slist1 = slist2;
```

The difference is that `slist2` now contains a copy of the 10 elements originally contained within `slist1`. Once again, if the size of the two containers are not the same, the container is resized to reflect the size of the container whose elements are being copied within it.

6.6.3 The Generic Algorithms

That's essentially all the operations a vector and deque container provide. Admittedly, that's a pretty thin interface, and omits some basic operations such as `find()`, `sort()`, `merge()`, and so on. Conceptually, the idea is to factor the operations common to all container types into a collection of generic algorithms that can be applied to all the container types, as well as the built-in array type. (The generic algorithms are discussed in detail in Chapter 12.) The generic algorithms are bound to a particular container through an iterator pair. For example, here is how we invoke the generic `find()` algorithm on a list, vector, and array of differing types:

```
#include <list>
#include <vector>

int ia[ 6 ] = { 0, 1, 2, 3, 4, 5 };
vector<string> svec;
list<double> dlist;

// the associated header file
#include <algorithm>

// find() returns an iterator to element, if found
// in case of array, returns a pointer ...
vector<string>::iterator viter;
list<double>::iterator liter;
```

```
int *pia;

pia = find( &ia[0], &ia[6], some_int_value );
liter = find( dlist.begin(), dlist.end(), some_double_value
);
viter = find( svec.begin(), svec.end(), some_string_value );
```

The list container type provides some additional operations, such as `merge()` and `sort()`, due to the fact that it does not support random access into its elements. We'll briefly look at these during our discussion of the associated generic algorithms in Chapter 12. Now let's turn to our text query system.

Exercise 6.11

Write a program that, given the following definitions,

```
int ia[] = { 1, 5, 34 };
int ia2[] = { 1, 2, 3 };
int ia3[] = { 6, 13, 21, 29, 38, 55, 67, 89 };
```

using the various insertion operations and the appropriate values of `ia2` and `ia3`, modify `ivec` to hold the sequence

```
{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 }
```

Exercise 6.12

Write a program that, given the following definitions,

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
list<int> ilist( ia, ia+11 );
```

using the single iterator form of `erase()`, remove all the odd numbered elements in `ilist`.

6.7 Storing Lines of Text

Our first task is to read in the text file against which our users wish to query. We'll need to retrieve the following information: each word, of course, but in addition the location of each word — that is, which line it is in and its position within that line. Moreover, we must preserve the text by line number in order to display the lines of text matching a query.

How will we retrieve each line of text? The standard library supports a `getline()` function declared as follows:

```
istream&
getline( istream &is, string str, char delimiter );
```

`getline()` reads the istream, inserting the characters, including whitespace, into the string object, until either (a) the delimiter is encountered, (b) the end-of-file occurs, or (c) the sequence of characters read equals the `max_size()` value of the string object, at which point the read operation fails.

Following each call of `getline()`, we'll insert `str` into the vector of string representing the text. Here is the general implementation[6] — we've factored it into a function we've named `retrieve_text()`. To add to the information collected, we've defined a pair of values to store the line number and length of the longest line. (The full program is listed in Section 6.14.)

```
// return value is a pointer to our string vector vec-
tor<string,allocator>*
retrieve_text()
{
  string file_name;

  cout << "please enter file name: ";
  cin  >> file_name;

  // open text file for input ...
  ifstream infile( file_name.c_str(), ios::in );
  if ( !infile ) {
       cerr << "oops! unable to open file "
            << file_name << " -- bailing out!\n";
       exit( -1 );
  }
  else cout << "\n";

  vector<string,allocator> *lines_of_text =
         new vector<string,allocator>;
  string textline;
```

[6] It is compiled under an implementation not supporting default template parameter values, and so we are required to explicitly provide an allocator:

```
vector< string, allocator > *lines_of_text;
```

In a fully-compliant Standard C++ implementation, we need only specify the element type:

```
vector< string > *lines_of_text;
```

```
    typedef pair<string::size_type, int> stats;
    stats maxline;
    int   linenum = 0;

    while (getline( infile, textline, '\n' ))
    {
      cout << "line read: " << textline << "\n";

      if ( maxline.first < textline.length() )
      {
           maxline.first = textline.length();
           maxline.second = linenum;
      }

      lines_of_text->push_back( textline );
      linenum++;
    }

    return lines_of_text;
}
```

The output of the program looks as follows (unfortunately, the lines wrap around due to the size of the text page — we've manually indented the second line to improve readability):

```
please enter file name: alice_emma

line read: Alice Emma has long flowing red hair. Her Daddy
says
line read: when the wind blows through her hair, it looks al-
most
           alive,
line read: like a fiery bird in flight. A beautiful fiery
bird, he
           tells her,
line read: magical but untamed. "Daddy, shush, there is no
such
           thing,"
line read: she tells him, at the same time wanting him to tell
her
           more.
```

```
line read: Shyly, she asks, "I mean, Daddy, is there?"

number of lines: 6
maximum length: 66
longest line: like a fiery bird in flight. A beautiful fiery
bird,
              he tells her,
```

Now that each text line is stored as a string, we need to break each line into its individual words. For each word, we'll first need to strip out punctuation. For example, this line from the *Anna Livia Plurrabelle* section of *Finnegans Wake*:

```
"For every tale there's a telling,
and that's the he and she of it."
```

yields the following individual strings with embedded punctuation

```
"For
there's
telling,
that's
it."
```

These need to become

```
For
there
telling
that
it
```

One could argue that

```
there's
```

should become

```
there is
```

but in fact we're going in the other direction: we're going to discard semantically neutral words such as *is*, *that*, *and*, *it*, *the*, and so on. And so for our active word set against which to query, of our line from Finnegans Wake, only

```
tale
telling
```

are entered. (We'll implement this using a word exclusion set, which we discuss in detail in the section below on the set container type.)

In addition to removing punctuation, we'll also need to strip out capitalization and provide some minimal handling of suffixing. Capitalization becomes a problem as in the following pair of text lines

```
Home is where the heart is.
A home is where they have to let you in.
```

Clearly, a query on `home` needs to find both entries.

Suffixing solves the more complicated problem of recognizing, for example, that dog and dogs represent the same noun, and that love, loves, loving, and loved represent the same verb.

Our purpose in the following sections is to revisit the standard library string class, exercising its extensive collection of string manipulation operations. Along the way, we will further evolve our text query system.

6.8 Finding a Substring

Our first task is to separate the string representing the line of text into its individual words. We'll do this by finding each embedded blank space. For example, given

```
Alice Emma has long flowing red hair.
```

by marking off the six embedded blank spaces, we can identify the seven substrings representing the actual words of the line of text. To do this, we use one of the `find()` functions supported by the string class.

The string class provides a collection of search functions each named as a variant of *find*. `find()` is the most straightforward instance: given a string, it returns either the index position of the first character of the matching substring, or else returns the special value

```
string::npos
```

indicating no match. For example,

```
#include <string>
#include <iostream>

int main() {
string name( "AnnaBelle" );
int pos = name.find( "Anna" );
if ( pos == string::npos )
     cout << "Anna not found!\n";
```

```
else cout << "Anna found at pos: " << pos << endl;
}
```

While the type of the index returned is almost always of type int, a more strictly portable and correct declaration uses

```
string::size_type
```

to hold the index value returned from `find()`. For example,

```
string::size_type pos = name.find( "Anna" );
```

`find()` does not provide us with the exact functionality we need; `find_first_of()`, however, does: `find_first_of()` returns the index position of the first character of the string that matches any character of a search string. For example, the following locates the first numeric value within a string:

```
#include <string>
#include <iostream>

int main() {
string numerics( "0123456789" );
string name( "r2d2" );

string::size_type pos = name.find_first_of( numerics );
cout << "found numeric at index: "
     << pos << "\telement is
     << name[pos] << endl;
}
```

In this example, `pos` is set to a value of 1 (the elements of a string, remember, are indexed beginning at 0).

This still does not do quite what we need, however. We need to find all occurrences in sequence, not just the first. We can do this by providing a second argument indicating the index position within the string to start our search. Here is a rewrite of our search of `"r2d2"`. It is still not quite right, however. Do you see what is wrong?

```
#include <string>
#include <iostream>

int main() {
string numerics( "0123456789" );
string name( "r2d2" );

string::size_type pos = 0;
```

```
// something wrong with implementation!
while (( pos = name.find_first_of( numerics, pos ))
!= string::npos )
cout << "found numeric at index: "
     << pos << "\telement is
     << name[pos] << endl;
}
```

`pos` begins the loop initialized to 0. The string is searched beginning at position 0. A match occurs at index 1. `pos` is assigned that value. Since it is not equal to `npos`, the body of the loop is executed. A second `find_first_of()` executes with `pos` set to 1. Oops! Position 1 matches a second, third, fourth time, and so on: we've programmed ourselves into an infinite loop. We need to increment `pos` one past the element found prior to each subsequent iteration of the loop:

```
// ok: corrected loop iteration
while (( pos = name.find_first_of( numerics, pos ))
!= string::npos )
{
cout << "found numeric at index: "
     << pos << "\telement is
     << name[pos] << endl;

// move 1 past element found
++pos;
}
```

To find the embedded white space within our line of text, we simply replace `numerics` with a string containing the possible white space characters we might encounter. However, if we are certain only a blank space is used, we can explicitly supply a single character. For example,

```
// program fragment
while (( pos = textline.find_first_of( ' ', pos ))
!= string::npos )
       // ...
```

To mark off the length of each word, we introduce a second positional object, as follows:

```
// program fragment
```

```
// pos: index 1 past word
// prev_pos: index beginning of word

string::size_type pos = 0, prev_pos = 0;

while (( pos = textline.find_first_of( ' ', pos ))
!= string::npos )
{
// do something with string
// now adjust positional markers
prev_pos = ++pos;
}
```

For each iteration of our loop, `prev_pos` indexes the beginning of the word, and `pos` holds the index *one past* the end of the word (the position of the space). The length of each identified word, then, is marked off by the expression

```
pos - prev_pos; // marks off length of word
```

Now that we've identified the word, we need to copy it, then tuck it away in a string vector. One strategy for copying the word is to loop through the textline from `prev_pos` to one less than `pos`, copying each character in turn, in effect extracting the substring marked off by the two indices. Rather than doing that ourselves, however, the `substr()` string operation does exactly that:

```
// program fragment

vector<string> words;
while (( pos = textline.find_first_of( ' ', pos ))
!= string::npos )
{
words.push_back(textline.substr(
                prev_pos, pos-prev_pos));
prev_pos = ++pos;
}
```

The `substr()` operation generates a copy of a substring of an existing string object. Its first argument indicates the start position within the string. The optional second argument indicates the length of the substring (if we leave off the second argument, the remainder of the string is copied).

There is one bug with our implementation: it fails to insert the last word of each line of text. Do you see why? Consider the line

```
seaspawn and seawrack
```

The first two words are marked off by a blank space. The position of the two blank spaces are returned in turn by the two invocations of `find_first_of()`. The third invocation, however, does not find a blank space; it sets `pos` to `string::npos`, terminating the loop. Processing of the final word, then, must follow termination of the loop.

Here is the full implementation, localized in a function we've named `separate_words()`. In addition to storing each word within a string vector, we've calculated the line and column position of each word. (We'll need this information later in support of positional text query.)

```
typedef pair<short,short> location;
typedef vector<location>  loc;
typedef vector<string>    text;
typedef pair<text*,loc*>  text_loc;

text_loc*
separate_words( const vector<string> *text_file )
{
    // words: holds the collection of individual words
    // locations: holds the associated line/col information
    vector<string> *words = new vector<string>;
    vector<location> *locations = new vector<location>;

    short line_pos = 0; // current line number

    // iterate through each line of text
    for ( ; line_pos < text_file->size(); ++line_pos )
    {
// textline: current line of text to process
// word_pos: current column position within textline
    short  word_pos = 0;
    string textline = (*text_file)[ line_pos ];

    string::size_type pos = 0, prev_pos = 0;

    while (( pos = textline.find_first_of( ' ', pos ))
 != string::npos )
         {
             // store a copy of the current word substring
             words->push_back(
                textline.substr( prev_pos, pos - prev_pos ));
```

```
                // store the line/col info as a pair
                locations->push_back(
                       make_pair( line_pos, word_pos ));

              // update position information for next iteration
                ++word_pos; ++pos; prev_pos = pos;
            }

            // now handle last word of line
            words->push_back(
                  textline.substr( prev_pos, pos - prev_pos ));

            locations->push_back(
                       make_pair( line_pos, word_pos ));
          }

          return new text_loc( words, locations );
}
```

The flow-of-control for our program thus far is as follows

```
int main()
{
    vector<string> *text_file = retrieve_text();
    text_loc *text_locations = separate_words( text_file );
    // ...
}
```

A trace of `separate_words()` on our input `text_file` looks as follows:

```
textline: Alice Emma has long flowing red hair. Her Daddy says

eol: 52 pos: 5 line: 0 word: 0 substring: Alice
eol: 52 pos: 10 line: 0 word: 1 substring: Emma
eol: 52 pos: 14 line: 0 word: 2 substring: has
eol: 52 pos: 19 line: 0 word: 3 substring: long
eol: 52 pos: 27 line: 0 word: 4 substring: flowing
eol: 52 pos: 31 line: 0 word: 5 substring: red
eol: 52 pos: 37 line: 0 word: 6 substring: hair.
eol: 52 pos: 41 line: 0 word: 7 substring: Her
eol: 52 pos: 47 line: 0 word: 8 substring: Daddy
last word on line substring: says
```

```
textline: when the wind blows through her hair, it looks al-
most alive,
eol: 60 pos: 4 line: 1 word: 0 substring: when
eol: 60 pos: 8 line: 1 word: 1 substring: the
eol: 60 pos: 13 line: 1 word: 2 substring: wind
eol: 60 pos: 19 line: 1 word: 3 substring: blows
eol: 60 pos: 27 line: 1 word: 4 substring: through
eol: 60 pos: 31 line: 1 word: 5 substring: her
eol: 60 pos: 37 line: 1 word: 6 substring: hair,
eol: 60 pos: 40 line: 1 word: 7 substring: it
eol: 60 pos: 46 line: 1 word: 8 substring: looks
eol: 60 pos: 53 line: 1 word: 9 substring: almost
last word on line substring: alive,

textline: like a fiery bird in flight. A beautiful fiery bird,
he tells her,
eol: 66 pos: 4 line: 2 word: 0 substring: like
eol: 66 pos: 6 line: 2 word: 1 substring: a
eol: 66 pos: 12 line: 2 word: 2 substring: fiery
eol: 66 pos: 17 line: 2 word: 3 substring: bird
eol: 66 pos: 20 line: 2 word: 4 substring: in
eol: 66 pos: 28 line: 2 word: 5 substring: flight.
eol: 66 pos: 30 line: 2 word: 6 substring: A
eol: 66 pos: 40 line: 2 word: 7 substring: beautiful
eol: 66 pos: 46 line: 2 word: 8 substring: fiery
eol: 66 pos: 52 line: 2 word: 9 substring: bird,
eol: 66 pos: 55 line: 2 word: 10 substring: he
eol: 66 pos: 61 line: 2 word: 11 substring: tells
last word on line substring: her,

textline: magical but untamed. "Daddy, shush, there is no such
thing,"
eol: 60 pos: 7 line: 3 word: 0 substring: magical
eol: 60 pos: 11 line: 3 word: 1 substring: but
eol: 60 pos: 20 line: 3 word: 2 substring: untamed.
eol: 60 pos: 28 line: 3 word: 3 substring: "Daddy,
eol: 60 pos: 35 line: 3 word: 4 substring: shush,
eol: 60 pos: 41 line: 3 word: 5 substring: there
eol: 60 pos: 44 line: 3 word: 6 substring: is
eol: 60 pos: 47 line: 3 word: 7 substring: no
eol: 60 pos: 52 line: 3 word: 8 substring: such
```

```
last word on line substring: thing,"

textline: she tells him, at the same time wanting him to tell
her more.
eol: 61 pos: 3 line: 4 word: 0 substring: she
eol: 61 pos: 9 line: 4 word: 1 substring: tells
eol: 61 pos: 14 line: 4 word: 2 substring: him,
eol: 61 pos: 17 line: 4 word: 3 substring: at
eol: 61 pos: 21 line: 4 word: 4 substring: the
eol: 61 pos: 26 line: 4 word: 5 substring: same
eol: 61 pos: 31 line: 4 word: 6 substring: time
eol: 61 pos: 39 line: 4 word: 7 substring: wanting
eol: 61 pos: 43 line: 4 word: 8 substring: him
eol: 61 pos: 46 line: 4 word: 9 substring: to
eol: 61 pos: 51 line: 4 word: 10 substring: tell
eol: 61 pos: 55 line: 4 word: 11 substring: her
last word on line substring: more.

textline: Shyly, she asks, "I mean, Daddy, is there?"
eol: 43 pos: 6 line: 5 word: 0 substring: Shyly,
eol: 43 pos: 10 line: 5 word: 1 substring: she
eol: 43 pos: 16 line: 5 word: 2 substring: asks,
eol: 43 pos: 19 line: 5 word: 3 substring: "I
eol: 43 pos: 25 line: 5 word: 4 substring: mean,
eol: 43 pos: 32 line: 5 word: 5 substring: Daddy,
eol: 43 pos: 35 line: 5 word: 6 substring: is
last word on line substring: there?"
```

Before we add to our set of text query routines, let's briefly cover the remaining search functions supported by the string class. In addition to `find()` and `find_first_of()`, the string class supports the additional find operations: `rfind()` searches for the last occurrence of the indicated substring. For example,

```
string river( "Mississippi" );

string::size_type first_pos = river.find( "is" );
string::size_type last_pos = river.rfind( "is" );
```

`find()` returns an index of 1, indicating the start of the first "*is*", while `rfind()` returns an index of 4, indicating the start of the last occurrence of "*is*".

`find_first_not_of()` searches for the first character of the string that does not match any element of the search string. For example, to find the first non-numeric

character of a string, we can write:

```
string elems( "0123456789" );
string dept_code( "03714p3" );

// returns index to the character 'p'
string::size_type pos = dept_code.find_first_not_of(elems);
```

`find_last_of()` searches for the last character of the string that matches any element of the search string. `find_last_not_of()` searches for the last character of the string that does not match any element of the search string. Each of these operations takes an optional second argument indicating the position within the string to begin searching.

Exercise 6.13

Write a program that, given the string

```
"ab2c3d7R4E6"
```

find (a) each numeric character then (b) each alphabet character first using `find_first_of()` then `find_first_not_of()`.

Exercise 6.14

Write a program that, given the string

```
string line1 = "We were her pride of 10 she named us --";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";

string sentence = line1 + line2 + line3;
```

count the number of words in sentence, and identify the largest and smallest words. If more than one word is either the largest or smallest, keep track of all.

6.9 Handling Punctuation

Now that we've separate each line of text into individual words, we need to remove any punctuation that may have stuck to the word. For example, the following line

```
magical but untamed. "Daddy, shush, there is no such thing,"
```

separates as follows:

```
magical
```

```
but
untamed.
"Daddy,
shush,
there
is
no
such
thing,"
```

How can we remove the unwanted punctuation? First, we'll define a string with all the punctuation elements we wish to remove:

```
string filt_elems( "\",.;:!?)(\\/" );
```

(The `\"` and `\\` sequences indicate that the quotation mark in the first sequence and the second slash in the second sequence are to be treated as literal elements within the quoted string.)

Next, we'll use the `find_first_of()` operation to find each matching element, if any, within our string:

```
while (( pos = word.find_first_of( filter, pos ))
!= string::npos )
```

Finally, we need to `erase()` the punctuation character from the string:

```
word.erase(pos,1);
```

The first argument to this version of the `erase()` operation indicates the position within the string to begin removing characters. An optional second argument indicates the number of characters to delete. In our example, we are deleting the one character located at `pos`. If we leave out the second argument, `erase()` removes all the characters from `pos` to the end of the string.

Here is the full listing of `filter_text()`. It takes two arguments, a pointer to our string vector containing the text, and a string object containing the elements to filter:

```
void
filter_text( vector<string> *words, string filter )
{
vector<string>::iterator iter = words->begin();
vector<string>::iterator iter_end = words->end();

// if no filter is provided by user, default to a minimal set
if ( ! filter.size() )
filter.insert( 0, "\"." );
```

```
while ( iter != iter_end ) {
string::size_type pos = 0;

// for each element found, erase it
while (( pos = (*iter).find_first_of( filter, pos ))
!= string::npos )
(*iter).erase(pos,1);
iter++;
}
}
```

Do you see why we do not increment `pos` with each iteration of the loop? That is, why the following is incorrect?

```
while (( pos = (*iter).find_first_of( filter, pos ))
!= string::npos ) {
(*iter).erase(pos,1);
++pos; // not correct ...
}
```

`pos` represents a position within the string. For example, given the string

```
thing,"
```

the first iteration of the loop assigns `pos` the value 5, the position of the comma. After we remove the comma, the string becomes

```
thing"
```

Position 5 is now the double quotation mark. If we had incremented `pos`, we would have failed to identify and remove this punctuation character.

Here is how we invoke `filter_text()` within our main program:

```
string filt_elems( "\",.;:!?)(\\/" );
filter_text( text_locations->first, filt_elems );
```

And, finally, here is a trace of the strings within our sample text in which one or more filter elements are found:

```
filter_text: hair.
found! : pos: 4.
after: hair

filter_text: hair,
found! : pos: 4,
```

```
after: hair

filter_text: alive,
found! : pos: 5,
after: alive

filter_text: flight.
found! : pos: 6.
after: flight

filter_text: bird,
found! : pos: 4,
after: bird

filter_text: her,
found! : pos: 3,
after: her

filter_text: untamed.
found! : pos: 7.
after: untamed

filter_text: "Daddy,
found! : pos: 0"
after: Daddy,
found! : pos: 5,
after: Daddy

filter_text: shush,
found! : pos: 5,
after: shush

filter_text: thing,"
found! : pos: 5,
after: thing"
found! : pos: 5"
after: thing

filter_text: him,
found! : pos: 3,
after: him
```

```
filter_text: more.
found! : pos: 4.
after: more

filter_text: Shyly,
found! : pos: 5,
after: Shyly

filter_text: asks,
found! : pos: 4,
after: asks

filter_text: "I
found! : pos: 0"
after: I

filter_text: mean,
found! : pos: 4,
after: mean

filter_text: Daddy,
found! : pos: 5,
after: Daddy

filter_text: there?"
found! : pos: 5?
after: there"
found! : pos: 5"
after: there
```

Exercise 6.15

Write a program that, given the string

```
"/.+(STL).*$1/"
```

erase all the characters except STL first using (a) `erase(pos,count)` and then (b) `erase(iter,iter)`.

Exercise 6.16

Write a program that, given the definitions

```
string sentence( "kind of" );
string s1( "whistle" );
string s2( "pixie" );
```

using the various insert string functions, provide `sentence` with the value:

```
"A whistling-dixie kind of walk."
```

6.10 A String By Any Other Format

One sort of nuisance detail of a text query system is the need to recognize the same word differing by tense, such as `cry`, `cries`, and `cried`, by number, such as `baby` and `babies`, and, more trivially, by capitalization, such as `home` and `Home`. The first two cases belong to the problem of word suffixing. While the general problem of suffixing is outside the scope of this text, the following small sample solution provides a good exercise of the string class operations.

Before we turn to suffixing, however, let's first solve the simpler case of capitalization. Rather than trying to be smart in recognizing special cases, we'll just replace all capital letters with their lowercase form. Our implementation looks as follows:

```
void
strip_caps( vector<string,allocator> *words )
{
vector<string,allocator>::iterator iter = words->begin();
vector<string,allocator>::iterator iter_end = words->end();

string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

while ( iter != iter_end ) {
string::size_type pos = 0;
while (( pos = (*iter).find_first_of( caps, pos )) !=
string::npos )
 (*iter)[ pos ] = tolower( (*iter)[pos] );
++iter;
}
}
```

The function

```
tolower( (*iter)[pos] );
```

is a Standard C library function that takes an uppercase character and returns its lowercase equivalent. To use it, we must include the header file

```
#include <ctype.h>
```

(This includes declarations of other functions such as `isalpha()`, `isdigit()`, `ispunct()`, `isspace()`, `toupper()`, and others. To find a full listing and discussion, see [PLAUGER92]. The Standard C++ Library defines both a ctype class that encapsulates the Standard C library functionality, as well as a set of non-member functions such as `toupper()`, `tolower()`, and so on. To use them, we must include the Standard C++ header file

```
#include <locale>
```

As of this writing, however, an implementation of this support is not available to us, and so we use the Standard C implementation.)

Suffixing is extremely difficult to rigorously implement; however, even less than perfect implementations yield significant improvements in the quality and size of our collection of words against which to query.

Our implementation only handles words that end in an `` `s` ``:

```
void
suffix_text( vector<string,allocator> *words )
{
vector<string,allocator>::iterator iter = words->begin();
vector<string,allocator>::iterator iter_end = words->end();

while ( iter != iter_end )
{
// if 3 or less characters, let it be
if ( (*iter).size() <= 3 ) { ++iter; continue; }
if ( (*iter)[ (*iter).size()-1 ] == 's' )
 suffix_s( *iter );

// additional suffix handling goes here such as
// ed, ing, ly

++iter;
}
}
```

A simple heuristic is to not bother with words under four characters. This saves us from dealing with `has`, `its`, `is`, and so on, but fails to catch, for example, `tv` and `tvs`

as being the same word.
If the word ends in "ies", as in babies and cries, we need to replace the "ies" with 'y':

```
string::size_type pos3 = word.size()-3;

string ies( "ies" );
if ( ! word.compare( pos3, 3, ies )) {
word.replace( pos3, 3, 1, 'y' );
return;
}
```

compare() returns 0 if the two strings being compared are equal. pos3 identifies the position within word to begin the comparison. The second argument, 3, in this case, indicates the length of the substring beginning at pos3. The third argument is the actual string against which to compare. (There are actually six versions of compare(). We look at the other versions briefly in the next section.)

replace() substitutes one or more characters within a string with one or more alternative characters. In our example, we replace the three character substring "ies" with a single repetition of the character 'y'. (There are ten overloaded instances of replace(). We'll revisit them briefly in the next section.)

Similarly, if the word ends in "ses", as in promises and crisises, we need to simply erase the ending "es":

```
string ses( "ses" );
if ( ! word.compare( pos3, 3, ses )) {
word.erase( pos3+1, 2 );
return;
}
```

If the word ends in "ous", as in oblivious, fulvous, and cretaceous, we do nothing. Similarly, if the word ends in "is", such as genesis, mimesis, and hepatitis, we do nothing. (This is not perfect, however. Kiwis, for example, requires that we drop the final 's'.) Also, if the word ends in "ius", as in genius, or in "ss", as in hiss, lateness, or less, we do nothing. To know whether or not to do nothing, we use a second form of compare():

```
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;

// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );
```

```
if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||    // ous
  ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||    // ius
  ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) || // ss
  ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))    // is
return;
```

Otherwise, we simply drop the final `'s'`:

```
// erase ending 's'
word.erase( pos3+2 );
```

Proper names, such as `Pythagoras`, `Brahms`, and the pre-Raphaelite painter `Burne-Jones`, fall outside the general rules. We'll handle them — well, actually leave that as an exercise for the reader, when we introduce the set associative container type. Before we turn to the map and set associative container types, we'd like to briefly cover some additional string operations in the next section.

Exercise 6.17

Our program leaves off the handling of suffixes ending in `ed`, as in `surprised`, `ly`, as in `surprisingly`, and `ing`, as in `surprising`. Add one of the following suffix handlers to the program: (a) `suffix_ed()`, (b) `suffix_ly()`, or (c) `suffix_ing()`.

6.11 Additional String Operations

A second form of `erase()` takes a pair of iterators marking off the range of characters to be deleted. For example, given the string

```
string name( "AnnaLiviaPlurabelle" );
```

let's produce a string *Annabelle*:

```
typedef string::size_type size_type;
size_type startPos = name.find( 'L' )
size_type endPos = name.find_last_of( 'a' );

name.erase( name.begin()+startPos,
name.begin()+endPos);
```

The character addressed by the second iterator is not part of the range of characters to be deleted.

Finally, a third form takes an iterator marking off a start position, and all the char-

acters from that position to the end of the string are removed. For example,

```
name.erase( name.begin()+4 );
```

leaves `name` with a string value of ``Anna''.

The `insert()` operation supports the insertion of additional characters into the string at the indicated position. Its general form is

```
string_object.insert( position, new_string );
```

where `position` indicates the location within `string_object` in which to insert `new_string`. `new_string` can be a string, C-style character string, or single character. For example,

```
string string_object( "Missisippi" );
string::size_type pos = string_object.find( "isi" );
string_object.insert( pos+1, 's' );
```

The `insert()` operation supports marking off a sub-portion of `new_string`. For example,

```
string new_string ( "AnnaBelle Lee" );
string_object += ' '; // append space

// find start and end position of new_string
pos = new_string.find( 'B' );
string::size_type posEnd = new_string.find( ' ' );

string_object.insert(
string_object.size(), // position within string_object
new_string, pos,      // start position within new_string
posEnd           // end position within new_string
)
```

`string_object` now contains the string ``Mississippi Belle''. If we wished to insert all of `new_string` starting at `pos`, we can leave off the `posEnd` value.

Given the following two strings

```
string s1( "Mississippi" );
string s2( "Annabelle" );
```

from out of these we'd like to create a third string with the value "`Miss Anna`". How might we do that?

One method is to make use of the `assign()` and `append()` string operations. These allow us to, in turn, copy and concatenate a portion of one string object to anoth-

er. For example,

```
string s3;

// copy first 4 characters of s1
s3.assign( s1, 4 );
```

`s3` now contains the value "`Miss`".

```
// concatenate a space
s3 += ' ';
```

`s3` now contains the value "`Miss `".

```
// concatenate the first 4 characters of s2
s3.append( s2, 4 );
```

`s3` now contains the value "`Miss Anna`". Alternatively, we can write this as

```
s3.assign( s1, 4 ).append( ' ' ).append( s2, 4 );
```

If we wish to extract a portion of the string that does not start at the beginning, we use an alternative form taking two integer values: a beginning position and a length. The position is counted beginning at 0. To extract "`belle`" from "`Annabelle`", for example, we specify a start position of 4, and a length of 5:

```
string beauty;

// assign "belle" to beauty
beauty.assign( s2, 4, 5 );
```

Alternatively, rather than provide a position and length, we can provide an iterator pair. For example,

```
// assign "belle" to beauty
beauty.assign( s2, begin()+4, s2.end() );
```

In the following example, we have two strings representing a current and a pending task. We need to periodically exchange them as we move from one project to the other and back again? For example,

```
string current_project( "C++ Primer, 3rd Edition" );
string pending_project( "Fantasia 2000, Firebird segment" );
```

The `swap()` operation exchanges the values of the two strings. Each invocation of

```
current_project.swap( pending_project );
```

exchanges the string values of the two objects.

Given the string

```
string first_novel( "V" );
```

the subscript

```
char ch = first_novel[ 1 ];
```

returns an undefined character value since the index is out of range: `first_novel` has a length of 1 indexed by the value 0. The subscript operator does not provide range checking, nor do we want it to on well-behaved code, such as the following:

```
int
elem_count( const string &word, char elem )
{
int occurs = 0;

// well-behaved: no need to check for out-of-bounds
for ( int ix=0; ix < word.size(); ++ix )
 if ( word[ ix ] == elem )
 ++occurs;
return occurs;
}
```

For potentially ill-defined code, however, such as

```
void
mumble( const string &st, int index )
{
// potential range error
char ch = st[ index ];

// ...
}
```

the alternative `at()` operation provides run-time range-checking of the index. If the index is valid, `at()` returns the associated character element the same as the subscript operator. If the index is invalid, however, `at()` raises the `out_of_range` exception:

```
void
mumble( const string &st, int index )
{
try {
```

```
char ch = st[ index ];
// ...
}
catch( std::out_of_range ) { ... }
// ...
}
```

Any two strings that are not equal have a lexigraphical — that is, dictionary — ordering. For example, given the following two strings

```
string cobol_program_crash( "abend" );
string cplus_program_crash( "abort" );
```

the `cobol_program_crash` string object is lexigraphically less than the `cplus_program_crash` string object through a comparison of the first non-equal character: `e` occurs before `o`, of course, in the English alphabet.

The `compare()` string operation provides for a lexigraphical comparision of two strings. Given

```
s1.compare( s2 );
```

`compare()` returns one of three possible values:

1. If `s1` is greater than `s2`, `compare()` returns a positive value.
2. If `s1` is less than `s2`, `compare()` returns a negative value.
3. If `s1` is equal to `s2`, `compare()` returns 0.

For example,

```
cobol_program_crash.compare(cplus_program_crash);
```

returns a negative value, while

```
cplus_program_crash.compare(cobol_program_crash);
```

returns a positive value. The string relational operators (<,>,<=,>=) provide an alternative shorthand notation for the `compare()` operation.

The overloaded set of six `compare()` operations allow us to mark off a substring of either one or both strings for comparison. Examples of this are presented in the previous section in the discussion of suffixing.

`replace()` provides us with ten ways of replacing one or more existing characters within a string with one or more alternative characters (the number of existing and replacement characters do not need to be equal). There are two primary formats to the `replace()` operation, with a subset of variations, based on the method of marking off the set of characters to be replaced. In one format, the first two arguments provide an

index to the start position of the character set, and a count of the number of characters to be replaced. In the second, a pair of iterators are passed in marking off the start position of the character set and one past the last character to be replaced. For example, here is an example of the first format:

```
string sentence(
   "An ADT provides both interface and implementation." );

string::size_type position = sentence.find_last_of( 'A' );
string::size_type length = 3;

// replace ADT with Abstract Data Type
sentence.replace( position, length, "Abstract Data Type" );
```

The first argument represents the start `position`, the second argument the `length` of the string beginning with `position`, so that a length of 3, not 2, represents the string ``ADT''. The third argument, of course, represents the new string. There are a number of variants with which to specify the new string. For example, this variant takes a string object rather than a C-style string

```
string new_str( "Abstract Data Type" );
sentence.replace( position, length, new_str );
```

while this variant in turn, admittedly inefficient, inserts a subportion of the new string marked off by a position and length.

```
#include <string>
typedef string::size_type size_type;

// get the position of the 3 words
size_type posA = new_str.find( 'A' );
size_type posD = new_str.find( 'D' );
size_type posT = new_str.find( 'T' );

// ok: replace T with "Type"
sentence.replace( position+2, 1, new_str, posT, 4 );

// ok: replace D with "Data "
sentence.replace( position+1, 1, new_str, posD, 5 );

// ok: replace A with "Abstract "
sentence.replace( position, 1, new_str, posA, 9 );
```

Another variant provides for the replacement of a substring with a single character repeated a specified number of time. For example,

```
string hmm( "Some celebrate Java as the successor to C++." );
string::size_type position = hmm.find( 'J' );

// ok: let's xxxx out Java
hmm.replace( position, 4, 'x', 4 );
```

There is one final variant we'd like to illustrate in which we use a pointer into an array of characters, and a length to mark off the new string. For example,

```
const char *lang = "EiffelAda95JavaModula3";
int index[] = { 0, 6, 11, 15, 22 };

string ahhem(
  "C++ is the language for today's power programmers." );

ahhem.replace(0, 3, lang+index[1], index[2]-index[1]);
```

Here is an example of the second format in which an iterator pair is used to mark off the substring targeted for replacement.

```
string sentence(
   "An ADT provides both interface and implementation." );

// points to the 'A' of ADT
string::iterator start = sentence.begin()+3;

// replace ADT with Abstract Data Type
sentence.replace( start, start+3, "Abstract Data Type" );
```

Four other variants allow for the replacement string to be a string object, to be a character inserted N times, to be a pair of iterators, or a C-style string in which N characters are used as the replacement character set.

This is all we wish to say about the string operations in our text. For more detailed or complete information, see the C++ Standard definition [ISO-C++98] (at the time of this writing, there is no preferred text on the Standard C++ Library).

Exercise 6.18

Write a program that, given the following two strings,

```
string quote1( "When lilacs last in the dooryard bloomed" );
```

```
string quote2( "The child is father to the man" );
```

using the `assign()` and `append()` operations, create the string

```
string sentence( "The child is in the dooryard" );
```

Exercise 6.19

Write a program that, given the strings,

```
string generic1( "Dear Ms Daisy:" );
string generic2( "MrsMsMissPeople" );
```

implement the function

```
string generate_salutation( string generic1,
                            string lastname,
                            string generic2,
                            string::size_type pos,
                            int length );
```

using the `replace()` operations, where `lastname` replaces `Daisy`, and `pos` indexes into `generic1` of `length` characters replacing `Ms`. For example, the following

```
string lastName( "AnnaP" );
string greetings =
      generate_salutation( generic1, lastName, generic2, 5,
4 );
```

returns the string

```
Dear Miss AnnaP:
```

6.12 Building A Text Location Map

In this section, we build a collection of line and column locations for each unique word in our text to introduce and explore the map associative container type. (In the following section, we build up a word exclusion set to introduce and explore the set associative container type.) In general, a set is most useful when we simply want to know whether a value has been seen or not, while a map is most useful when we wish to store (and possibly modify) an associated value. In both cases, the elements are stored in an ordered relationship to support efficient storage and retrieval.

In a map, also known as an *associative array*, we provide a key/value pair: the key serves as an index into the map, and the value serves as the actual data to be stored

and retrieved. In our program example, each string object serves as a key, the vector of line and column locations as the value. To access the location vector, we index into the map using the subscript operator. For example,

```
string query( "pickle" );
vector< location > *locat;

// returns location<vector>* associated with "pickle"
locat = text_map[ query ];
```

The map's key type — string in our example — serves as the index. The associated `location<vector>*` value is returned.

To use a map, we must include its associated header file

```
#include <map>
```

The two primary activities in the use of a map (and set) are to (1) populate it with elements and (2) query it as to the presence or absence of an element. In the next subsection, we look at how we define and insert key/value pairs. In the subsection subsequent to that, we look at how we discover whether or not an element is present and, if so, how we retrieve its value.

6.12.1 Defining and Populating a Map

To define a map object, we must minimally indicate a key and value type. For example,

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

defines a map object `word_count` that is indexed by a string and that holds an associated `int` value. Similarly,

```
class employee;
map<int,employee*> personnel;
```

defines a map object `personnel` that is indexed by an `int` (it represents a unique employee number) and that holds an associated pointer to an instance of class employee.

For our text query system, our map declaration looks as follows:

```
typedef pair<short,short> location;
typedef vector<location> loc;

map<string,loc*> text_map;
```

(Because the compilers available to us at the time of this writing do not support de-

fault arguments for template parameters, in practice we must provide the following expanded definition:

```
map<string,loc*,    // key,value pair
    less<string>,   // relational operator for ordering
    allocator>      // default memory allocator
text_map;
```

By default, the associative container types are ordered using the less-than operator. We can always override that, however, indicating an alternative relational operator (see Section 12.3 on function objects).

Once the map is defined, the next step is to populate it with the key/value element pairs. Intuitively, what we'd like to write is the following:

```
#include <map>
#include <string>
map<string,int> word_count;

word_count[ string("Anna") ] = 1;
word_count[ string("Danny") ] = 1;
word_count[ string("Beth") ] = 1;

// and so on ...
```

When we write

```
word_count[ string("Anna") ] = 1;
```

the following steps take place:

1. An unnamed temporary string object initialized with "Anna" is constructed and passed to the subscript operator associated with the map class.
2. The word_count map is searched for the entry "Anna". The entry is not found.
3. A new key/value pair is inserted into word_count. The key, of course, is a string object holding the value "Anna". The value, however, is not 1, but 0.
4. The insertion done, the value is then assigned 1.

When a key is inserted into a map through the subscript operator, the associated value is initialized to the default value of the underlying element type. The default value of the built-in arithmetic types is 0.

In effect, using the subscript operator to initialize a map to a collection of elements causes each value to be initialized to a default value, then assigned the explicit value. If the elements are class objects for which the default initialization and assignment are computationally significant, the performance of our programs can be impacted, al-

though the program's correctness remains unaffected.

The preferred, if syntactically more intimidating insertion method for a single element is the following:

```
// the preferred single element insertion method
word_count.insert(
  map<string,int>::
       value_type( string("Anna"), 1 )
);
```

The map defines a type `value_type` which represents its associated key/value pair. The effect of the lines

```
map<string,int>::
     value_type( string("Anna"), 1 )
```

is to create a pair object that is then directly inserted within the map. For readability, we can use a typedef, as follows:

```
typedef map<string,int>::value_type valType;
```

Using this, our insertion appears somewhat less complicated:

```
word_count.insert( valType( string("Anna"), 1 ));
```

To insert a range of key/value elements, the `insert()` method taking a pair of iterators can be used. For example,

```
map< string, int > word_count;
// ... fill it up

map< string, int > word_count_two;

// insert a copy of all the key/value pairs
word_count_two.insert( word_count.begin(),
            word_count.end() );
```

In this example, the same effect could be achieved by initializing the second map object to the first, as follows:

```
// initialize with a copy of all the key/value pairs
map< string, int > word_count_two( word_count );
```

Let's walk through how we might build up our text map. `separate_words()`, discussed in Section 6.8, creates two vectors: (1) a string vector of each word of the text, and (2) a location vector holding a line and column pair of values. For each word ele-

ment in the string vector, the equivalent element of the location vector provides the line and column information for that word. The string vector, that is, provides the collection of key values for our text map. The location vector provides the associated collection of values.

`separate_words()` returns these two vectors in a pair object holding a pointer to each. The argument to our `build_word_map()` function is this pair object. The return value is the text location map — or, rather, a pointer to it:

```
// typedefs to make declarations easier
typedef pair<short,short>          location;
typedef vector<location >          loc;
typedef vector<string >            text;
typedef pair<text*,loc*>           text_loc;

extern map< string, loc* >*
           build_word_map( const text_loc *text_locations );
```

Our first prepatory steps are to (1) allocate the empty map from the free store, and (2) separate out the string and location vectors from the pair argument passed in as the argument:

```
map<string,loc*> *word_map = new map< string, loc* >;

vector<string>   *text_words = text_locations->first;
vector<location> *text_locs  = text_locations->second;
```

Next, we need to iterate across the two vectors in parallel. There are two cases to consider:

1. The word does not yet exist within the map. In this case, we need to insert the key/value pair.
2. The word has already been inserted. In this case, we need to update the location vector of the entry with the additional line and column information.

Here is our implementation:

```
register int elem_cnt = text_words->size();
for ( int ix = 0; ix < elem_cnt; ++ix )
{
   string textword = ( *text_words )[ ix ];

   // exclusion strategies
   // less than 3 character or in exclusion set
   if ( textword.size() < 3 ||
```

```
            exclusion_set.count( textword ))
                continue;

        // determine whether the word is present
        // if count() returns 0, not present -- add it
        if ( ! word_map->count((*text_words)[ix] ))
        {
           loc *ploc = new vector<location>;
           ploc->push_back( (*text_locs)[ix] );
           word_map->insert( value_type( (*text_words)[ix], ploc
));
        }
        else
          // update the location vector of the entry

    (*word_map)[(*text_words)[ix]]>push_back((*text_locs)[ix]);
    }
```

The syntactically complex expression

```
(*word_map)[(*text_words)[ix]]>push_back((*text_locs)[ix]);
```

is perhaps more easily understand if we decompose it into its individual components:

```
// get the word to update
string word = (*text_words)[ix];

// get the location vector
vector<location> *ploc = (*word_map)[ word ];

// get the line and column pair
location loc = (*text_locs)[ix];

// insert the new location pair
ploc->push_back(loc);
```

The remaining syntactic complexity is due to our manipulating pointers to vectors rather than the vectors themselves. To directly apply the subscript operator, we cannot write

```
string word = text_words[ix]; // error
```

but must first dereference our pointer:

```
string word = (*text_words)[ix]; // ok
```

Finally, `build_word_map()` returns our built-in map:

```
string word = (*text_words)[ix]; // ok
```

Here is how we might invoke it within our `main()` function:

```
int main()
{
// read in and separate the text
vector<string,allocator> *text_file = retrieve_text();
text_loc *text_locations = separate_words( text_file);

// process the words
// ...

// build up the word/location map and invite query
map<string,loc*,less<string>,allocator>
*text_map = build_word_map( text_locations );

// ...
}
```

6.12.2 Finding and Retrieving a Map Element

The subscript operator provides the simplest method of retrieving a value. For example,

```
// map<string,int> word_count;
int count = word_count[ "wrinkles" ];
```

This behaves satisfactorily, however, only when there is an instance of the key present within the map. If the instance is not present, the use of the subscript operator causes an instance to be inserted. In this example, the key/value pair

```
string( "wrinkles" ), 0
```

is inserted into `word_count` and `count` is initialized to 0.

There are two map operations to discover if the key element is present without its absence causing an instance to be inserted:

1. `count(keyValue):count()` returns the number of occurrences of `keyValue` within a map. (For a map, of course, the return value can only be 0 or 1. For a multimap,) If non-zero, we can safely use the subscript operator. For example,

   ```
   int count = 0;
   ```

```
if ( word_count.count( "wrinkles" ))
     count = word_count[ "wrinkles" ];
```

1. `find(keyValue):find()` returns an iterator to the instance within the map, if it is present, or else an iterator equal to `end()` if the instance is not present. For example,

```
int count = 0;
map<string,int>::iterator it = word_count.find( "wrinkles" );
if ( it != word_count.end() )
     count = (*it).second;
```

The iterator to an element of a map addresses a pair object in which `first` holds the key and `second` holds the value (we'll look at this again in the next subsection).

6.12.3 Iterating Across a Map

Now that we've built our map, we'd like to print out its contents. We can do this by iterating across the elements marked off by the `begin()` and `end()` pair of iterators. Here is our function, `display_map_text()`, that does just that.

```
void
display_map_text( map<string,loc*> *text_map )
{
typedef map<string,loc*> tmap;
tmap::iterator iter = text_map->begin(),
iter_end = text_map->end();

while ( iter != iter_end )
{
cout << "word: " << (*iter).first << " (";

int           loc_cnt   = 0;
loc          *text_locs = (*iter).second;
loc::iterator liter     = text_locs->begin(),
  liter_end = text_locs->end();

while ( liter != liter_end )
{
if ( loc_cnt )
   cout << ",";
else ++loc_cnt;
```

```
cout << "(" << (*liter).first
     << "," << (*liter).second << ")";

++liter;
}

cout << ")\n";
++iter;
}

cout << endl;
}
```

If the map is without elements, there is no point bothering to invoke our display function. One method of discovering if the map is empty is to invoke `size()`:

```
if ( text_map->size() )
display_map_text( text_map );
```

But rather than unnecessarily counting all the elements, we can more directly invoke `empty()`:

```
if ( ! text_map->empty() )
display_map_text( text_map );
```

6.12.4 A Word Transformation Map

Here is a small program to illustrate building up, searching, and iterating across a map. We use two maps in our program. Our word transformation map holds two elements of type string. The key represents a word requiring special handling; the value represents the transformation we should apply whenever we encounter the word. For simplicity, we hard-code the pairs of map entries (as an exercise, you may wish to generalize the program to read in pairs of word transformations from either standard input or a user-specified file). Our statistics map stores usage statistics of actual transformations performed. Here is our program.

```
#include <map>
#include <vector>
#include <iostream>
#include <string>

int main()
{
    map< string, string > trans_map;
```

```
    typedef map< string, string >::value_type valType;

    // a first expedient: hand-code the transformation map
    trans_map.insert( valType( "gratz", "grateful" ));
    trans_map.insert( valType( "'em",   "them"     ));
    trans_map.insert( valType( "cuz",   "because"  ));
    trans_map.insert( valType( "nah",   "no"       ));
    trans_map.insert( valType( "sez",   "says"     ));
    trans_map.insert( valType( "tanx",  "thanks"   ));
    trans_map.insert( valType( "wuz",   "was"      ));
    trans_map.insert( valType( "pos",   "suppose"  ));

    // ok: let's display it
    map< string,string >::iterator it;

    cout << "Here is our transformation map: \n\n";
    for ( it = trans_map.begin();
          it != trans_map.end(); ++it )
          cout << "key: "   << (*it).first << "\t"
               << "value: " << (*it).second << "\n";
    cout << "\n\n";

    // a second expedient: hand-code the text ...
    string textarray[14]={ "nah", "I", "sez", "tanx", "cuz",
"I",
        "wuz", "pos", "to", "not", "cuz", "I", "wuz", "gratz"
};

    vector< string > text( textarray, textarray+14 );
    vector< string >::iterator iter;

    // ok: let's display it
    cout << "Here is our original string vector:\n\n";
    int cnt = 1;
    for ( iter = text.begin(); iter != text.end(); ++iter,
++cnt )
          cout <<  *iter << ( cnt % 8 ? " " : "\n" );

    cout << "\n\n\n";

    // a map to hold statistics -- build up dynamically
```

```
    map< string,int > stats;
    typedef map< string,int >::value_type statsValType;

    // ok: the actual mapwork -- heart of the program
    for ( iter = text.begin(); iter != text.end(); ++iter )
       if (( it = trans_map.find( *iter )) != trans_map.end()
)
          {
                if ( stats.count( *iter ))
                     stats[ *iter ] += 1;
               else stats.insert( statsValType( *iter, 1 ));
                *iter = (*it).second;
          }

    // ok: display the transformed vector
    cout << "Here is our transformed string vector:\n\n";
    cnt = 1;
    for ( iter = text.begin(); iter != text.end(); ++iter,
++cnt )
          cout <<  *iter << ( cnt % 8 ? " " : "\n" );
    cout << "\n\n\n";

    // ok: now iterate over the statistics map
    cout << "Finally, here are our statistics:\n\n";
    map<string,int,less<string>,allocator>::iterator siter;

   for ( siter = stats.begin(); siter != stats.end(); ++siter
)
          cout << (*siter).first     << " "
               << "was transformed "
               << (*siter).second
               << ((*siter).second == 1
                              ? " time\n" : " times\n" );
}
```

When executed, the program generates the following output:

```
Here is our transformation map:

key: 'em         value: them
key: cuz         value: because
key: gratz       value: grateful
```

```
key: nah          value: no
key: pos          value: suppose
key: sez          value: says
key: tanx         value: thanks
key: wuz          value: was


Here is our original string vector:

nah I sez tanx cuz I wuz pos
to not cuz I wuz gratz


Here is our transformed string vector:

no I says thanks because I was suppose
to not because I was grateful


Finally, here are our statistics:

cuz was transformed 2 times
gratz was transformed 1 time
nah was transformed 1 time
pos was transformed 1 time
sez was transformed 1 time
tanx was transformed 1 time
wuz was transformed 2 times
```

6.12.5 Erasing Elements from a Map

There are three variants of the `erase()` operation for removing elements from a map. To erase a single element, we pass `erase()` either a key-value or an iterator. To remove a sequence of elements, we pass `erase()` an iterator pair. For example, if we wish to allow our user to remove elements from `text_map`, we might do the following:

```
string removal_word;
cout << "type in word to remove: ";
cin >> removal_word;
if ( text_map->erase( removal_word ))
cout << "ok: " << removal_word << " removed\n";
```

```
else cout << "oops: " << removal_word << " not found!\n";
```

Alternatively, before we attempt to erase the word, we can check to see if it is present:

```
map<string,loc*>::iterator where;
where = text_map.find( removal_word );

if ( where == text_map->end() )
cout << "oops: " << removal_word << " not found!\n";
else {
text_map->erase( where );
cout << "ok: " << removal_word << " removed!\n";
}
```

In our implementation of `text_map`, we store multiple locations associated with each word. This complicates the storage and retrieval of the actual location values. An alternative implementation is to insert a word entry for each location. A map, however, holds only a single instance of a key value. To provide multiple entries of the same key, we must use a multimap. Section 6.15, below, looks at the multimap associative container type.

Exercise 6.20

Define a map for which the index is the family surname, and the key is a vector of the children's names. Populate the map with at least six entries. Test it by (a) supporting user queries based on a surname, (b) adding a child to one family, triplets to another, and (c) printing out all the map entries.

Exercise 6.21

Extend the map of the previous exercise by having the vector store a pair of strings: the child's name and birthday. Revise the Exercise 6.20 implementation to support the new pair vector. Test your modified test program to verify its correctness.

Exercise 6.22

List at least three possible applications in which map type might be of use. Write out the definition of each map, and indicate how the elements are likely to be inserted and retrieved.

6.13 Building A Word Exclusion Set

A map consists of a key/value pair, such as an address and phone number keyed to an individual's name. In contrast, a set is simply a collection of key values. For example, a business might define a set `bad_checks` consisting of names of individuals who have issued bad checks to the company over the past two years. A set is most useful when we simply want to know whether a value is present or not. Before accepting our check, for example, that business may wish to query `bad_checks` to see if either of our names are present.

For our text query system, we build a word exclusion set of semantically neutral words such as *the*, *and*, *into*, *with*, *but*, and so on. (While this provides significant improvement in the quality of our word index, it does result in our inability to locate the first line of Hamlet's famous speech, *"To be or not to be"*.) Prior to entering a word into our map, we check whether it is present within the word exclusion set. If it is, we do not enter it into the map.

6.13.1 Defining and Populating a Set

To define or make use of the set associative container type, we must include its associated header file

```
#include <set>
```

Here is our definition of our word exclusion set object:

```
set<string> exclusion_set;
```

Individual elements are added to the set using the insert operation. For example,

```
exclusion_set.insert( "the" );
exclusion_set.insert( "and" );
```

Alternatively, we can insert a range of elements by providing a pair of iterators to `insert()`. For example, our text query system allows the user to specify a file of words to exclude from our map. If the user chooses not to supply a file, we fill the set with a default collection of words:

```
typedef set< string >::difference_type diff_type;
set< string > exclusion_set;

ifstream infile( "exclusion_set" );
if ( !infile )
{
static string default_excluded_words[25] = {
```

```
"the","and","but","that","then","are","been",
"can","can't","cannot","could","did","for",
"had","have","him","his","her","its","into",
"were","which","when","with","would"
};

cerr << "warning! unable to open word exclusion file! -- "
   << "using default set\n";

copy( default_excluded_words, default_excluded_words+25,
 inserter( exclusion_set, exclusion_set.begin() ));
}
else {
istream_iterator< string, diff_type > input_set( infile ),
eos;
copy( input_set, eos, inserter( exclusion_set,
 exclusion_set.begin() ));
}
```

This code fragment introduces two elements that we have not seen as yet: `difference_type` and the `inserter` class. `difference_type` is the type of the result of subtracting two iterators of our string set. The istream_iterator uses this as one of its parameters.

`copy()`, of course, is one of the generic algorithms (we discuss them in detail in Chapter 12 and the Appendix). It's first two arguments are either iterators or pointers marking off the range of elements to copy. The third argument is either an iterator or pointer to the beginning of the container into which to copy the elements.

The problem is that `copy()` expects a container of a size equal to or greater than the number of elements to be copied. This is because `copy()` assigns each element in turn; it does not insert the elements. The associative containers, however, do not support the preassignment of a size. In order to copy the elements into our exclusion set, we must somehow have `copy()` insert rather than assign each element. The inserter class accomplishes just that. (It is discussed in detail in Section 12.4.)

6.13.2 Searching for an Element

The two operations to query a set object as to whether a value is present are `find()` and `count()`. `find()` returns an iterator addressing the element found, if present, or else an iterator equal to `end()` indicating the element is not present. `count()`

returns 1 if the element is found; it returns 0 if the element is not present. Within `build_word_map()`, we add a test of exclusion_set prior to entering the word within our map:

```
if (exclusion_set.count( textword ))
continue;
// ok: enter word into map
```

6.13.3 Iterating Across a Set

To exercise our word/locations map, we implemented a small function to permit single word queries (support for the full query language is presented in Chapter 17). If the word is found, we wish to display each line within which the word occurs. A word, however, might occur multiple times within a single line, as in

```
tomorrow and tomorrow and tomorrow
```

and we want to make sure that we display this line only once.

One strategy for maintaining only one instance of each line the word occurs in is to use a set, as we do in the following code fragment:

```
// retrieve pointer to location vector
loc *ploc = (*text_map)[ query_text ];

// iterate across location entry pairs
// insert each line value into set
set< short > occurrence_lines;
loc::iterator liter = ploc->begin(), liter_end = ploc->end();

while ( liter != liter_end ) {
occurrence_lines.insert(occurrence_lines.end(), (*li-
ter).first);
++liter;
}
```

A set may only contain a single instance of each key value. `occurrence_lines`, therefore, is guaranteed to contain one instance of each line within which the word occurs. To display these lines of text, we simply iterate across the set:

```
register int size = occurrence_lines.size();
cout << "\n" << query_text
  << " occurs " << size
  << (size == 1 ? " time:" : " times:")
  << "\n\n";
```

```
set< short >::iterator it=occurrence_lines.begin();
for ( ; it != occurrence_lines.end(); ++it ) {
int line = *it;

cout << "\t( line "
  << line + 1 << " ) "
  << (*text_file)[line] << endl;
}
```

(The full implementation of `query_text()` is presented in the following section.)

A set supports the operations `size()`, `empty()`, and `erase()` the same as does the map type described in the previous section. In addition, the generic algorithms provide a collection of set specific functions such as `set_union()` and `set_difference()`. (We'll make use of these in Chapter 17 in support of our query language.)

Exercise 6.23

Add an exclusion set of handling words in which the trailing `'s'` should not be removed, but for which there exists no general rule. For example, three words to place in this set are the proper names Pythagoras, Brahms, and Burne_Jones. Fold the use of this exclusion set into the `suffix_s()` function of Section 6.10.

Exercise 6.24

Define a vector of books you'd like to read within the next virtual six months, and a set of titles that you've read. Write a program that chooses a next book for you to read from the vector provided you have not as yet read it. When it returns the selected title to you, it should enter the title in the set of books read. If in fact you end up putting it aside, provide support for removing the title from the set of books read. At the end of our virtual six months, print out the set of books read and those books that were not read.

6.14 The Complete Program

This section presents the full working program developed within this chapter with two modifications: (1) rather than preserve the procedural organization of separate data structures and functions, we have introduced a TextQuery class to encapsulate both (we'll look at this use of a class in more detail in subsequent chapters), and (2) we are presenting the text as it was modified to compile under the currently available implementations. The iostream library reflects a prestandard implementation, for

example. Templates do not support default arguments for template parameter. To have the program run on your current system, you may need to modify this or that declaration.

```
// standard library header files
#include <algorithm>
#include <string>
#include <vector>
#include <utility>
#include <map>
#include <set>

// prestandard iostream header file
#include <fstream.h>

// Standard C header files
#include <stddef.h>
#include <ctype.h>

// typedefs to make declarations easier
typedef pair<short,short>           location;
typedef vector<location,allocator>  loc;
typedef vector<string,allocator>    text;
typedef pair<text*,loc*>            text_loc;

class TextQuery {
public:
  TextQuery() { memset( this, 0, sizeof( TextQuery )); }

  static void filter_elements( string felems )
{ filt_elems = felems; }

  void query_text();
  void display_map_text();
  void display_text_locations();
  void doit() {

private:

  void retrieve_text();
  void separate_words();
  void filter_text();
```

```
    void strip_caps();
    void suffix_text();
    void suffix_s( string& );
    void build_word_map();

private:
    vector<string,allocator>     *lines_of_text;
    text_loc                     *text_locations;
    map<string,loc*,less<string>,allocator> *word_map;
    static string                filt_elems;
};

string TextQuery::filt_elems( "\",.;:!?)(\\/" );

int main()
{
    TextQuery tq;
    tq.doit();
    tq.query_text();
    tq.display_map_text();
}

void
TextQuery::
retrieve_text()
{
    string file_name;

    cout << "please enter file name: ";
    cin  >> file_name;

    ifstream infile( file_name.c_str(), ios::in );
    if ( !infile ) {
         cerr << "oops! unable to open file "
              << file_name << " -- bailing out!\n";
         exit( -1 );
    }
    else cout << "\n";

    lines_of_text = new vector<string,allocator>;
    string textline;
```

```
  while ( getline( infile, textline, '\n' ))
          lines_of_text->push_back( textline );
}

void
TextQuery::
separate_words()
{
  vector<string,allocator> *words = new vector<string,allo-
cator>;
  vector<location,allocator> *locations =
         new vector<location,allocator>;

  for ( short line_pos = 0; line_pos < lines_of_text->size();
        line_pos++ )
  {
        short  word_pos = 0;
        string textline = (*lines_of_text)[ line_pos ];

        string::size_type eol = textline.length();
        string::size_type pos = 0, prev_pos = 0;

        while (( pos = textline.find_first_of( ' ', pos ))
                    != string::npos )
        {
                  words->push_back(
                      textline.substr( prev_pos, pos -
prev_pos ));
                  locations->push_back(
                      make_pair( line_pos, word_pos ));

                  word_pos++; pos++; prev_pos = pos;
        }

        words->push_back(
            textline.substr( prev_pos, pos - prev_pos ));

        locations->push_back(make_pair(line_pos,word_pos));
  }
```

```
  text_locations = new text_loc( words, locations );
}

void
TextQuery::
filter_text()
{
  if ( filt_elems.empty() )
       return;

  vector<string,allocator> *words = text_locations->first;

  vector<string,allocator>::iterator iter = words->begin();
  vector<string,allocator>::iterator iter_end = words-
>end();

  while ( iter != iter_end )
  {
          string::size_type pos = 0;
          while (( pos = (*iter).find_first_of( filt_elems,
pos ))
                       != string::npos )
                  (*iter).erase(pos,1);
    iter++;
  }
}

void
TextQuery::
suffix_text()
{
  vector<string,allocator> *words = text_locations->first;

  vector<string,allocator>::iterator iter = words->begin();
  vector<string,allocator>::iterator iter_end = words-
>end();

  while ( iter != iter_end )
  {
    // if 3 or less characters, let it be
    if ( (*iter).size() <= 3 )
```

```
            { iter++; continue; }

        if ( (*iter)[ (*iter).size()-1 ] == 's' )
              suffix_s( *iter );

        // additional suffix handling goes here ...

        iter++;
    }
}

void
TextQuery::
suffix_s( string &word )
{
    string::size_type spos = 0;
    string::size_type pos3 = word.size()-3;

    // "ous", "ss", "is", "ius"
    string suffixes( "oussisius" );

    if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||
         ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||
         ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) ||
         ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))
             return;

    string ies( "ies" );
    if ( ! word.compare( pos3, 3, ies ))
    {
         word.replace( pos3, 3, 1, 'y' );
         return;
    }

    string ses( "ses" );
    if ( ! word.compare( pos3, 3, ses ))
    {
         word.erase( pos3+1, 2 );
         return;
    }
```

```
    // erase ending 's'
    word.erase( pos3+2 );

    // watch out for "'s"
    if ( word[ pos3+1 ] == '\'' )
         word.erase( pos3+1 );
}

void
TextQuery::
strip_caps()
{
    vector<string,allocator> *words = text_locations->first;

    vector<string,allocator>::iterator iter = words->begin();
    vector<string,allocator>::iterator iter_end = words-
>end();

    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

    while ( iter != iter_end ) {
            string::size_type pos = 0;
           while (( pos = (*iter).find_first_of( caps, pos ))
                       != string::npos )
                  (*iter)[ pos ] = tolower( (*iter)[pos] );
            ++iter;
    }
}


void
TextQuery::
build_word_map()
{
    word_map = new map< string, loc*, less<string>, allocator
>;

    typedef map<string,loc*,less<string>,alloca-
tor>::value_type
             value_type;
```

```
  typedef set<string,less<string>,alloca-
tor>::difference_type
          diff_type;

  set<string,less<string>,allocator> exclusion_set;

  ifstream infile( "exclusion_set" );
  if ( !infile )
  {
    static string default_excluded_words[25] = {
      "the","and","but","that","then","are","been",
      "can","can't","cannot","could","did","for",
      "had","have","him","his","her","its","into",
      "were","which","when","with","would"
    };

    cerr << "warning! unable to open word exclusion file! -
_ "
         << "using default set\n";

    copy( default_excluded_words,
default_excluded_words+25,
          inserter( exclusion_set, exclusion_set.begin() ));
  }
  else {
    istream_iterator< string, diff_type > input_set( infile
),
                                            eos;
    copy( input_set, eos,
          inserter( exclusion_set, exclusion_set.begin() ));
  }

  // iterate through the the words, entering the key/pair

  vector<string,allocator> *text_words  = text_locations-
>first;
  vector<location,allocator> *text_locs = text_locations-
>second;

  register int elem_cnt = text_words->size();
  for ( int ix = 0; ix < elem_cnt; ++ix )
```

```
    {
      string textword = ( *text_words )[ ix ];

      // exclusion strategies
      // less than 3 character or in exclusion set
      if ( textword.size() < 3 ||
           exclusion_set.count( textword ))
                continue;

      if ( ! word_map->count((*text_words)[ix] ))
      {  // not present, add it:
         loc *ploc = new vector<location,allocator>;
         ploc->push_back( (*text_locs)[ix] );
        word_map->insert( value_type( (*text_words)[ix], ploc
));
      }
      else (*word_map)[(*text_words)[ix]]->
                  push_back( (*text_locs)[ix] );
    }
}

void
TextQuery::
query_text()
{
   string query_text;

   do {
      cout << "enter a word against which to search the
text.\n"
           << "to quit, enter a single character ==>  ";
      cin  >> query_text;

      if ( query_text.size() < 2 ) break;

      string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
      string::size_type pos = 0;
      while (( pos = query_text.find_first_of( caps, pos ))
                  != string::npos )
               query_text[ pos ] = tolower( query_text[pos] );
```

```
    // if we index into map, query_text is entered, if absent
    // not at all what we should wish for ...

    if ( !word_map->count( query_text )) {
         cout << "\nSorry. There are no entries for "
              << query_text << ".\n\n";
         continue;
    }

    loc *ploc = (*word_map)[ query_text ];

    set<short,less<short>,allocator> occurrence_lines;
    loc::iterator liter = ploc->begin(),
                  liter_end = ploc->end();

    while ( liter != liter_end ) {
      occurrence_lines.insert(
         occurrence_lines.end(), (*liter).first);
      ++liter;
    }

    register int size = occurrence_lines.size();
    cout << "\n" << query_text
         << " occurs " << size
         << (size == 1 ? " time:" : " times:")
         << "\n\n";

    set<short,less<short>,allocator>::iterator
              it=occurrence_lines.begin();
    for ( ; it != occurrence_lines.end(); ++it ) {
          int line = *it;

          cout << "\t( line "
               // don't confound user with
               // text lines starting at 0
               << line + 1 << " ) "
               << (*lines_of_text)[line] << endl;
    }

    cout << endl;
}
```

```
  while ( ! query_text.empty() );
  cout << "Ok, bye!\n";
}

void
TextQuery::
display_map_text()
{
  typedef map<string,loc*,less<string>,allocator> map_text;
  map_text::iterator iter = word_map->begin(),
                     iter_end = word_map->end();

  while ( iter != iter_end ) {
    cerr << "word: " << (*iter).first << " (";

    int           loc_cnt = 0;
    loc          *text_locs = (*iter).second;
    loc::iterator liter     = text_locs->begin(),
                  liter_end = text_locs->end();

    while ( liter != liter_end )
    {
      if ( loc_cnt )
           cerr << ",";
      else ++loc_cnt;

      cerr << "(" << (*liter).first
           << "," << (*liter).second << ")";

      ++liter;
    }

    cerr << ")\n";
    ++iter;
  }

  cerr << endl;
}

void
TextQuery::
```

```
display_text_locations()
{
  vector<string,allocator>   *text_words     =
text_locations->first;
  vector<location,allocator> *text_locs      =
text_locations->second;

  register int elem_cnt = text_words->size();

  if ( elem_cnt != text_locs->size() )
  {
    cerr << "oops! internal error: word and position vectors
"
         << "are of unequal size\n"
         << "words: " << elem_cnt << " "
         << "locs: "  << text_locs->size()
         << " -- bailing out!\n";
    exit( -2 );
  }

  for ( int ix = 0; ix < elem_cnt; ix++ )
  {
    cout << "word: " << (*text_words)[ ix ] << "\t"
         << "location: ("
         << (*text_locs)[ix].first  << ","
         << (*text_locs)[ix].second << ")"
         << "\n";
  }

  cout << endl;
}
```

Exercise 6.25

We have not yet introduced the special inserter iterator, which we need to use to populate the exclusion word set. (It is introduced in Chapter 12.) Can you guess why it is necessary?

```
set<string> exclusion_set;
ifstream    infile( "exclusion_set" );
// ...
copy( default_excluded_words, default_excluded_words+25,
```

```
    inserter( exclusion_set, exclusion_set.begin() ));
```

Exercise 6.26

Our current implementation reflects a procedural solution — that is, a collection of global functions operating on an independent set of unencapsulated data structures. An alternative solution is to wrap the functions and data structures in a TextQuery class. Do you think this would or would not improve our program? Why?

Exercise 6.27

In this version of the program, the user is prompted for the text file to be handled. A more convenient implementation would allow the user to specify the file on the program command line — we'll see how to support command line arguments to a program in the next chapter. What other command line options should our program support?

6.15 Multimap/Multiset

Both a map and set may contain only a single instance of each key. The multiset and multimap allow for multiple occurrences of a key to be stored. A phone directory, for example, may wish to provide a separate listing for each phone number associated with an individual. A listing of available texts by an author may wish to provide a separate listing for each title. Or a word text may wish to provide a separate location pair for each occurrence of a word within the text. To use a multimap or multiset, the associated map or set header file needs to be included:

```
#include <map>
multimap< key_type, value_type > multimapName;

// indexed by string, holding list< string >
multimap< string, list< string > > synonyms;

#include <set>
multiset< type > multisetName;
```

For either a multimap or multiset, one iteration strategy is to use a combination of the iterator returned by `find()` (it points to the first instance) and the value returned by `count()`. (This works because the instances are guaranteed to occur contiguously within the container.) For example,

```
#include <map>
#include <string>
```

```
void code_fragment()
{
  multimap< string, string > authors;
  string search_item( "Alain de Botton" );
  // ...
  int number = authors.count( search_item );
  multimap< string,string >::iterator iter;

  iter = authors.find( search_item );
  for ( int cnt = 0; cnt < number; ++cnt, ++iter )
    do_something( *iter );

  // ...
}
```

An alternative, somewhat more elegant strategy, is to use the pair of iterator values returned by the special multiset and multimap operation, `equal_range()`. If the value is present, the first iterator points to the first instance of the value and the second iterator points one past the last instance of the value if the last instance is the last element of the multiset, the second iterator is set equal to `end()`. For example,

```
#include <map>
#include <string>
#include <utility>

void code_fragment()
{
  multimap< string, string > authors;
  // ...
  string search_item( "Haruki Murakami" );

  while ( cin && cin >> search_item )
    switch ( authors.count( search_item ))
    {
      // none present, go to the next item
      case 0:
        break;

      // single item. ordinary find()
      case 1: {
        multimap< string,string >::iterator iter;
```

```
                iter = authors.find( search_item );
                // do something with element
                break;

            // multiple entries present ...
            default:
            {
                typedef multimap< string,string >::iterator iter-
ator;
                pair< iterator, iterator > pos;

                // pos.first addresses first occurrence
                // pos.second addresses position in which
                //     value no longer occurs
                pos = authors.equal_range( search_item );
                for ( ; pos.first != pos.second; pos.first++ )
                      // do something with each element
        }
      }
    }
```

Insertion and deletion is the same as for the simpler map and set associative container types. `equal_range()` is useful for providing the iterator pair necessary to mark off the range of multiple elements to be erased. For example,

```
#include <multimap>
#include <string>

typedef multimap< string, string >::iterator iterator;
pair< iterator, iterator > pos;
string search_item( "Kazuo Ishiguro" );

// authors is a multimap<string,string>
// this is equivalent to
// authors.erase( search_item );
pos = authors.equal_range( search_item );
authors.erase( pos.first, pos.second );
```

Insertion adds an additional element each time. For example,

```
typedef multimap<string,string>::value_type valType;
multimap<string,string> authors;
```

```
// introduces a first key under Barth
authors.insert( valType(
  string("Barth, John"),
  string("Sot-Weed Factor")));

// introduces a second key under Barth
authors.insert( valType(
  string("Barth, John"),
  string("Lost in the Funhouse")));
```

One constraint on access of an element of a multimap is that the subscript operator is not supported. For example, the following

```
authors[ "Barth, John" ]; // error: multimap
```

results in a compile-time error.

Exercise 6.28

Reimplement the text query program of Section 6.14 to make use of a multimap in which each location is separately entered. What are the performance and design characteristics of the two solutions? Which do you feel is the preferred design solution? Why?

6.16 Stack

In Section 4.6, in illustrating the increment and decrement operators, recall, we implemented a stack abstraction. In general, stacks provide a powerful solution to the problem of maintaining a current state when multiple nesting states can occur dynamically during the course of program execution. Because a stack is such an important data abstraction, the standard C++ library provides a class implementation. To use it, we must include its associated header file:

```
#include <stack>
```

The stack provided by the standard library is implemented slightly differently from ours, in that the access and removal of the top element are separated into, respectively, a `top()` and `pop()` pair of operations. The full set of operations supported by

the stack container are the following:

empty()	returns true if the stack is empty; false otherwise.
size()	returns a count of the number of elements on the stack.
pop()	removes, but does not return, the top element from the stack.
top()	returns, but does not remove, the top element on the stack.
push(item)	places a new top element on the stack.

The following program exercises this set of five stack operations:

```
#include <stack>
#include <iostream>

int main()
{
const int ia_size = 10;
int ia[ia_size ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// fill up the stack
int ix = 0;
stack< int > intStack;
for ( ; ix < ia_size; ++ix )
  intStack.push( ia[ ix ] );

int error_cnt = 0;
if ( intStack.size() != ia_size ) {
  cerr << "oops! invalid intStack size: "
    << intStack.size()
    << "\t expected: " << ia_size << endl;
  ++error_cnt;
}

int value;
while ( intStack.empty() == false )
{
     // read the top element of the stack
     value = intStack.top();
```

```
        if ( value != --ix ) {
           cerr << "oops! expected " << ix
                << " received " << value
                << endl;
           ++error_cnt;
        }

        // pop the top element, and repeat
        intStack.pop();
    }

    cerr << "Our program ran with "
      << error_cnt << " errors!" << endl;
    }
```

The declaration

```
stack< int > intStack;
```

declares `intStack` to be an empty stack of integer elements. The stack type is spoken of as a *container adaptor* because it imposes the stack abstraction on an underlying container collection. By default, the stack is implemented using the deque container type, since a deque provides for the efficient insertion and deletion at the front of a container, which a vector does not. If we should wish to override the default, we can define a stack object providing an explicit sequence container type as a second argument, for example,

```
stack< int, list<int> > intStack;
```

The elements of the stack are entered by value; that is, each object is copied onto the underlying container. For large or complex class objects, this may prove overly expensive, particularly if we are only reading the elements. An alternative storage strategy is to define a stack of pointers. For example,

```
#include <stack>

class NurbSurface { /* mumble */ };
stack< NurbSurface* > surf_Stack;
```

Two stacks of the same type can be compared for equality, inequality, less-than, greater-than, less-than-equal, and greater-than-equal relationships provided that the underlying element type supports the equality and less-than operators. For these operations, the elements are compared in turn. The first pair of non-equal elements de-

termines the less or greater-than relationship.

We illustrate the program use of a stack in Section 17.X in our support of complex user text queries such as

```
Civil && ( War || Rights )
```

6.17 Queue and Priority Queue

A queue abstraction exhibits a *FIFO* storage and retrieval policy — that is, first in, first out. Objects entering the queue are placed in the back. The next object retrieved is taken from the front of the queue. There are two flavors of queues provided by the standard library, the FIFO queue, which we will speak of simply as a queue, and a priority queue.

A priority queue allows the user to establish a priority among the items held within the queue. Rather than placing a newly entered item at the back of the queue, the item is placed ahead of all those items with a lower priority. The user defining the priority queue determines how that priority is to be decided. A real-world example of a priority queue is that of the line to check-in luggage at an airport. Those whose flight is going to leave within the next 15 minutes are generally moved to the front of the line in order that they can finish the check-in process prior to their plane taking off. A programming example of a priority queue is the scheduler of an operating system determining which, of a number of waiting processes, should execute next.

In order to make use of either a queue or priority_queue, the associated header file must be included:

```
#include <queue>
```

The full set of operations supported by both the queue and priority_queue containers are the following:

empty()	returns true if the queue is empty; false otherwise.
size()	returns a count of the number of elements on the queue.
pop()	removes, but does not return, the front element from the queue. In the case of the priority_queue, the front element represents the element with the highest priority.
front()	returns, but does not remove, the front element on the queue. This can only be applied to a queue.
back()	returns, but does not remove, the back element on the queue. This can only be applied to a queue.
top()	returns, but does not remove, the highest priority element of the priority_queue. This can only be applied to a priority_queue.
push(item)	places a new element at the end of the queue. In the case of a priority_queue, order the item based on its priority.

An ordering is imposed on the elements of a priority_queue such that the elements are arranged from largest to smallest, where largest is equivalent to having the highest priority. By default, the prioritization of the elements is carried out by the less-than operator associated with the underlying element type. If we should wish to override the default less-than operator, we can explicitly provide a function or function object to be used for ordering the priority_queue elements (Section 12.3 explains and illustrates this further).

6.18 Revisiting our iStack class

The iStack class presented in Section 4.15 is constrained in two regards:

1. It only supports a single type — that of type `int`. We'd prefer to support all element types. We can do this by transitioning our implementation to support a general template Stack class.
2. It is fixed length. This is problematic in two regards: (a) our stack can become full and therefore unusable, and (b) to avoid having the stack become full, we end up allocating on average considerably too much space. The solution is to support a dynamically growing stack. We can do this by making direct use of the dynamic support provided by the underlying vector object.

Before we begin, here is our original iStack class definition:

```
#include <vector>
```

```
class iStack {
public:
    iStack( int capacity )
            : _stack( capacity ), _top( 0 ) {};

    bool pop( int &value );
    bool push( int value );

    bool full();
    bool empty();
    void display();

    int size();

private:
    int _top;
    vector< int > _stack;
};
```

Let's first transition the class to support dynamic allocation. Essentially, this means we must insert and remove elements rather than index into a fixed size vector. The data member `_top` is no longer necessary; the use of `push_back()` and `pop_back()` manages the top element automatically. Here is our revised implementations of `pop()` and `push()`:

```
bool iStack::pop( int &top_value )
{
if ( empty() )
     return false;
top_value = _stack.pop_back();
return true;
}

bool iStack::push( int value )
{
if ( full() )
     return false;
_stack.push_back( value );
return true;
}
```

`empty()`, `size()`, and `full()` must also be reimplemented — more tightly coupled in this version to the underlying vector:

```
inline bool iStack::empty(){ return _stack.empty(); }
inline bool iStack::size(){ return _stack.size(); }
inline bool iStack::full(){
       return _stack.max_size() == _stack.size(); }
```

`display()` requires a slight modification to remove its use of `_top` as an end condition to the for-loop:

```
void iStack::display()
{
cout << "( " << size() << " )( bot: ";
for ( int ix = 0; ix < size(); ++ix )
      cout << _stack[ ix ] << " ";
cout << " :top )\n";
}
```

Our only significant design decision has to do with our revised iStack constructor. Strictly speaking, our constructor no longer needs to do anything, and the following null constructor is sufficient for our reimplemented iStack class:

```
inline iStack::iStack() {}
```

It is not sufficient for our users, however. As of this point, we have exactly preserved the original interface, and so no existing user code needs to be rewritten. To be fully compatible with our original iStack interface, we must retain a single argument constructor, although we don't wish to require it as our original implementation did. Our modified interface accepts but does not require a single argument of type `int`:

```
class iStack {
public:
iStack( int capacity = 0 );
// ...
};
```

What do we do with the argument, if present? We'll use it to set the vector's capacity:

```
inline iStack::iStack( int capacity )
{
if ( capacity )
     _stack.reserve( capacity );
};
```

The transition from a non-template to template class is even easier, in part due to the underlying vector object already belonging to a template class. Here is our revised class declaration:

```
#include <vector>

template <class elemType>
class Stack {
public:
    Stack( int capacity=0 );

    bool pop( elemType &value );
    bool push( elemType value );

    bool full();
    bool empty();
    void display();

    int size();

private:
    vector< elemType > _stack;
};
```

To preserve compatibility with existing programs making use of our earlier iStack class implementation, we provide the following `typedef`:

```
typedef Stack<int> iStack;
```

We leave the revision of the member operations as an exercise.

Exercise 6.29

Reimplement the `peek()` function (Exercise 4.24 of Section 4.15) for our dynamic iStack class.

Exercise 6.30

Provide the revised member operations for our template Stack class. Run the test program of Section 4.15 against the new implementation.

Exercise 6.31

Using the model of the List class in the last section of Chapter 5, encapsulate our tem-

plate Stack class in the `Primer_Third_Edition` namespace.