CHAPTER 22

STL Containers

Objectives

- To know the relationships among containers, iterators, and algorithms (§22.2).
- To distinguish sequence containers, associative containers, and container adapters (§22.2).
- To distinguish containers **`vector`**, **`deque`**, **`list`**, **`set`**, **`multiset`**, **`map,`** **`multimap`**, **`stack`**, **`queue`**, and **`priority_queue`** (§22.2).
- To use common features of containers (§22.2).
- To access elements in a container using iterators (§22.3).
- To distinguish iterator types: input, output, forward, bidirectional, and random-access (§22.3.1).
- To manipulate iterators using operators (§22.3.2).
- To obtain iterators from containers and know the type of interators supported by containers (§22.3.3).
- To perform input and output using **`istream_iterator`** and **`ostream_iterator`** (§22.3.4).
- To declare auto variable for automatic type inference (§22.4).
- To store, retrieve, and process elements in sequence containers: **`vector`**, **`deque`**, and **`list`** (§22.5).
- To store, retrieve, and process elements in associative containers: **`set`**, **`multiset`**, **`map`**, and **`multimap`** (§22.6).
- To store, retrieve, and process elements in container adapters: **`stack`**, **`queue`**, and **`priority_queue`** (§22.7).

22.1 Introduction

Key Point: *STL provides a standard library for classic data structures.*

Chapter 20, “Linked Lists, Queues, and Priority Queues,” and Chapter 21, “Binary Search Trees,” introduced several data structures such as linked lists, stacks, queues, heaps, and priority queues. These popular data structures are used widely in many applications. C++ provides a library known as the *Standard Template Library* (*STL*) for these and many other useful data structures. So, you can use them without having to reinvent the wheel. One example that you have learned is the **`vector`** class, which was introduced in §12.6, “The C++ Vector Class.” This chapter introduces the STL, and you will learn how to use the classes in it to simplify application development.

22.2 STL Basics

Key Point: *STL contains three main components: containers, iterators, and algorithms.*

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard, based on their research in generic programming in collaboration with David Musser. It is a collection of libraries written in C++. The classes and functions in the STL are template classes and template functions.

The STL contains three main components:

- ***Containers***: Classes in the STL are container classes. A container object such as a vector is used to store a collection of data, often referred to as *elements*.
- ***Iterators***: The STL container classes make extensive use of iterators, which are objects that facilitate traversing through the elements in a container. Iterators are like built-in pointers that provide a convenient way to access and manipulate the elements in a container.
- ***Algorithms***: Algorithms are used in the functions to manipulate data, such as sorting, searching, and comparing elements. About 80 algorithms are implemented in the STL. Most of them use iterators to access the elements in the container. STL algorithms will be introduced in Chapter 23.

The STL containers can be classified into three categories:

- ***Sequence containers***: The sequence containers (also known as sequential containers) represent linear data structures. The three sequence containers are **`vector`**, **`list`**, and **`deque`** (pronounced deck).

- ***Associative containers***: Associative containers are nonlinear containers that can quickly locate elements stored in them. Such containers can store sets of values or *key/value* pairs. The four associative containers are **`set`**, **`multiset`**, **`map`**, and **`multimap`**.
- ***Container adapters***: Container adapters are constrained versions of sequence containers. They are adapted from sequence containers for handling special cases. The three container adapters are **`stack`**, **`queue`**, and **`priority_queue`**.

Table 22.1 summarizes the container classes and their header files.

Table 22.1

Container Classes

	STL Container	Header File	Applications	
Sequence containers	vector	<vector>	For direct access to any element, quick insertion, and deletion at the end of the vector.	First-class containers
	deque	<deque>	For direct access to any element, quick insertion, and deletion at the front and end of the deque.	
	list	<list>	For rapid insertion and deletion anywhere.	
Associative containers	set	<set>	For direct lookup, no duplicated elements.	
	multiset	<set>	Same as set, except that duplicated elements allowed.	
	map	<map>	Key/value pair mapping, no duplicates allowed, and quick lookup using the key.	
	multimap	<map>	Same as map, except that keys may be duplicated.	
Container adapters	stack	<stack>	Last-in, first-out container.	
	queue	<queue>	First-in, first-out container.	
	priority_queue	<queue>	The highest-priority element is removed first.	

All STL containers share some common features and functions. For example, each container has a no-arg constructor, a copy constructor, a destructor, and so on. Table 22.2 lists the common functions for all containers, and Table 22.3 lists the common functions for the sequence containers and associative containers. These two containers also are known as the *first-class containers*.

Table 22.2

Common Functions to All Containers

Functions	Description
non-arg constructor	Constructs an empty container.
constructor with args	In addition to the no-arg constructor, every container has several constructors with args.
copy constructor	Creates a container by copying the elements from an existing container of the same type.
destructor	Performs cleanup after the container is destroyed.
empty()	Returns true if there are no elements in the container.
size()	Returns the number of elements in the container.
operator=	Copies one container to another.
Relational operators (<, <=, >, >=, ==, and !=)	The elements in the two containers are compared sequentially to determine the relation.

Table 22.3

Common Functions to First-Class Containers

STL Functions	Description
c1.swap(c2)	Swaps the elements of two containers c1 and c2.
c.max_size()	Returns the maximum number of elements a container can hold.
c.clear()	Erases all elements from the container.
c.begin()	Returns an iterator to the first element in the container.
c.end()	Returns an iterator that refers to the position after the end of the container.
c.rbegin()	Returns an iterator to the last element in the container for processing elements in reverse order.
c.rend()	Returns an iterator that refers to the position before the first element in the container.
c.erase(beg, end)	Erases the elements in the container from beg to end-1. Both beg and end are iterators.

Listing 22.1 gives a simple example that demonstrates how to create a **vector**, **list**, **deque**, **set**, **multiset**, **stack**, and **queue**.

Listing 22.1 SimpleSTLDemo.cpp

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

```cpp
#include <stack>
#include <queue>
using namespace std;

int main()
{
  vector<int> vector1, vector2;
  list<int> list1, list2;
  deque<int> deque1, deque2;
  set<int> set1, set2;
  multiset<int> multiset1, multiset2;
  stack<int> stack1, stack2;
  queue<int> queue1, queue2;

  cout << "Vector: " << endl;
  vector1.push_back(1);
  vector1.push_back(2);
  vector2.push_back(30);
  cout << "size of vector1: " << vector1.size() << endl;
  cout << "size of vector2: " << vector2.size() << endl;
  cout << "maximum size of vector1: " << vector1.max_size() << endl;
  cout << "maximum size of vector2: " << vector1.max_size() << endl;
  vector1.swap(vector2);
  cout << "size of vector1: " << vector1.size() << endl;
  cout << "size of vector2: " << vector2.size() << endl;
  cout << "vector1 < vector2? " << (vector1 < vector2)
    << endl << endl;

  cout << "List: " << endl;
  list1.push_back(1);
  list1.push_back(2);
  list2.push_back(30);
  cout << "size of list1: " << list1.size() << endl;
  cout << "size of list2: " << list2.size() << endl;
  cout << "maximum size of list1: " << list1.max_size() << endl;
  cout << "maximum size of list2: " << list2.max_size() << endl;
  list1.swap(list2);
  cout << "size of list1: " << list1.size() << endl;
  cout << "size of list2: " << list2.size() << endl;
  cout << "list1 < list2? " << (list1 < list2) << endl << endl;

  cout << "Deque: " << endl;
  deque1.push_back(1);
  deque1.push_back(2);
  deque2.push_back(30);
  cout << "size of deque1: " << deque1.size() << endl;
  cout << "size of deque2: " << deque2.size() << endl;
  cout << "maximum size of deque1: " << deque1.max_size() << endl;
  cout << "maximum size of deque2: " << deque2.max_size() << endl;
  list1.swap(list2);
  cout << "size of deque1: " << deque1.size() << endl;
  cout << "size of deque2: " << deque2.size() << endl;
  cout << "deque1 < deque2? " << (deque1 < deque2) << endl << endl;

  cout << "Set: " << endl;
  set1.insert(1);
  set1.insert(1);
  set1.insert(2);
  set2.insert(30);
  cout << "size of set1: " << set1.size() << endl;
  cout << "size of set2: " << set2.size() << endl;
  cout << "maximum size of set1: " << set1.max_size() << endl;
  cout << "maximum size of set2: " << set2.max_size() << endl;
  set1.swap(set2);
  cout << "size of set1: " << set1.size() << endl;
  cout << "size of set2: " << set2.size() << endl;
  cout << "set1 < set2? " << (set1 < set2) << endl << endl;

  cout << "Multiset: " << endl;
  multiset1.insert(1);
  multiset1.insert(1);
  multiset1.insert(2);
  multiset2.insert(30);
  cout << "size of multiset1: " << multiset1.size() << endl;
  cout << "size of multiset2: " << multiset2.size() << endl;
  cout << "maximum size of multiset1: " <<
        multiset1.max_size() << endl;
  cout << "maximum size of multiset2: " <<
        multiset2.max_size() << endl;
  multiset1.swap(multiset2);
  cout << "size of multiset1: " << multiset1.size() << endl;
  cout << "size of multiset2: " << multiset2.size() << endl;
  cout << "multiset1 < multiset2? " <<
        (multiset1 < multiset2) << endl << endl;

  cout << "Stack: " << endl;
  stack1.push(1);
  stack1.push(1);
  stack1.push(2);
  stack2.push(30);
  cout << "size of stack1: " << stack1.size() << endl;
  cout << "size of stack2: " << stack2.size() << endl;
  cout << "stack1 < stack2? " << (stack1 < stack2) << endl << endl;

  cout << "Queue: " << endl;
  queue1.push(1);
  queue1.push(1);
  queue1.push(2);
  queue2.push(30);
  cout << "size of queue1: " << queue1.size() << endl;
  cout << "size of queue2: " << queue2.size() << endl;
  cout << "queue1 < queue2? " << (queue1 < queue2) << endl << endl;

  return 0;
}
```

Sample output

```
Vector:
size of vector1: 2
size of vector2: 1
maximum size of vector1: 1073741823
maximum size of vector2: 1073741823
size of vector1: 1
size of vector2: 2
```

```
vector1 < vector2? 0

List:
size of list1: 2
size of list2: 1
maximum size of list1: 4294967295
maximum size of list2: 4294967295
size of list1: 1
size of list2: 2
list1 < list2? 0

Deque:
size of deque1: 2
size of deque2: 1
maximum size of deque1: 4294967295
maximum size of deque2: 4294967295
size of deque1: 1
size of deque2: 2
deque1 < deque2? 0

Set:
size of set1: 2
size of set2: 1
maximum size of set1: 4294967295
maximum size of set2: 4294967295
size of set1: 1
size of set2: 2
set1 < set2? 0

Multiset:
size of multiset1: 3
size of multiset2: 1
maximum size of multiset1: 4294967295
maximum size of multiset2: 4294967295
size of multiset1: 1
size of multiset2: 3
multiset1 < multiset2? 0

Stack:
size of stack1: 3
size of stack2: 1
stack1 < stack2? 1

Queue:
size of queue1: 3
size of queue2: 1
queue1 < queue2? 1
```

Each container has a no-arg constructor. The program creates vectors, lists, deques, sets, multisets, stacks, and queues in lines 12–18 using the container's no-arg constructors.

The program uses the **`push_back(element)`** function to append an element into a vector, list, and deque in lines 21–23, 35–37, and 48–50; the **`insert(element)`** function to insert an element to a set and multiset in lines 61–64 and 75–78; and the `push(element)` function to push an element into a stack and queue in lines 92–95 and 101–104.

Integer `1` is inserted into `set1` twice in lines 61–62. Since a set does not allow duplicate elements, `set1` contains {`1`, `2`} after `2` is inserted into `set1` in line 63. A multiset allows duplicates, so `multiset1` contains {`1`, `1`, `2`} after `1`, `1`, and `2` are inserted into `multiset1` in lines 75–77.

All containers support the relational operators. The program compares two containers of the same type in lines 31, 45, 58, and 72.

Check Point

22.1 What are the three main components of the STL? What are the relationships among them?

22.2 What are the three types of containers? What are they used for?

22.3 Does C++ define a base class for all containers?

22.4 Which of the following are the common features for all containers?

a. Each container has a no-arg constructor.

b. Each container has a copy constructor.

c. Each container has the `empty()` function to check whether a container is empty.

d. Each container has the `size()` function to return the number of elements in the container.

e. Each container supports the relational operators (`<`, `<=`, `>`, `>=`, `==`, and `!=`).

22.5 What is a first-class container?

22.6 Which containers use iterators?

22.7 Which of the following are the common features for all first-class containers?

a. Each first-class container has the `swap` function.

b. Each first-class container has the `max_size()` function.

c. Each first-class container has the `clear()` function.

d. Each first-class container has the `erase` function.

e. Each first-class container has the `add` function.

22.3 STL Iterators

Key Point: *STL iterators provide a unformed way for traversing the elements in a container.*

Iterators are used extensively in the first-class containers for accessing and manipulating the elements. As you already have seen in Table 22.3, several functions (e.g., **begin()** and **end()**) in the first-class containers are related to iterators. §20.5, "Iterators," presented examples on how to implement iterators in a container. You'll find it helpful to review §20.5 before reading this section.

The **begin()** function returns the iterator that points to the first element in a container, and the **end()** function returns the iterator that represents a position past the last element in a container, as pictured in Figure 22.1.

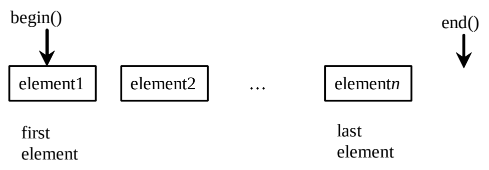

Figure 22.1

end() *represents a position past the last element.*

Typically, you cycle through all the elements in a container using the following loop:

```
for (iterator p = c.begin(); p != c.end(); p++)
{
  processing *p; // *p is the current element
}
```

Each container has its own iterator type. The abstraction hides the detailed implementation and provides a uniform way for using iterators on all containers. Iterators are used in the same way in all containers, so if you know how to use iterators with one container class, you can apply this to all other containers.

Listing 22.2 demonstrates using iterators in a vector and a set.

Listing 22.2 IteratorDemo.cpp

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

```
 4  using namespace std;
 5
 6  int main()
 7  {
 8    vector<int> intVector;
 9    intVector.push_back(10);
10    intVector.push_back(40);
11    intVector.push_back(50);
12    intVector.push_back(20);
13    intVector.push_back(30);
14
15    vector<int>::iterator p1;
16    cout << "Tranverse the vector: ";
17    for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
18    {
19      cout << *p1 << " ";
20    }
21
22    set<int> intSet;
23    intSet.insert(10);
24    intSet.insert(40);
25    intSet.insert(50);
26    intSet.insert(20);
27    intSet.insert(30);
28
29    set<int>::iterator p2;
30    cout << "\nTranverse the set: ";
31    for (p2 = intSet.begin(); p2 != intSet.end(); p2++)
32    {
33      cout << *p2 << " ";
34    }
35    cout << endl;
36
37    return 0;
38  }
```

Sample output

```
      Traverse the vector: 10 40 50 20 30
      Traverse the set: 10 20 30 40 50
```

The program creates a vector for **`int`** values (line 8), appends five numbers (lines 9–13), and traverses the vector using an iterator (lines 15–20).

An iterator **`p1`** is declared in line 15:

```
        vector<int>::iterator p1;
```

Every container has its own iterator type. Here **`vector<int>::iterator`** denotes the iterator type in the **`vector<int>`** class.

The expression (line 17)

```
p1 = intVector.begin();
```

obtains the iterator that points to the first element in the vector **`intVector`** and assigns the iterator to **`p1`**.

The expression (line 17)

```
p1 != intVector.end();
```

checks whether **`p1`** has passed the last element in the container.

The expression (line 17)

```
p1++
```

advances the iterator to the next element.

The expression (line 19)

```
*p1
```

returns the element pointed by **`p1`**.

Similarly, the program creates a set for **`int`** values (line 22), inserts five numbers (lines 23–27), and traverses the set using an iterator (line 29–34). Note that the elements in a set are sorted, so the program displays **`10`**, **`20`**, **`30`**, **`40`**, and **`50`** in the sample output.

From this example, you can see that an iterator functions like a pointer. An iterator variable points to an element in the container. You use the increment operator (**`p++`**) to move the iterator to the next element and use the dereference operator (**`*p`**) to access the element.

22.3.1 Type of Iterators

Each container has its own iterator type. Iterators can be classified into five categories:

- ***Input iterators***: An input iterator is used for reading an element from a container. It can move only in a forward direction one element at a time.
- ***Output iterators***: An output iterator is used for writing an element to a container. It can move only in a forward direction one element at a time.

- ***Forward iterator***: A forward iterator combines all the functionalities of input and output iterators to support both read and write operations.
- ***Bidirectional iterator***: A bidirectional iterator is a forward iterator with the capability of moving backward. It can be moved freely back or forth one element at a time.
- ***Random-access iterator***: A random-access iterator is a bidirectional iterator that can access any element in any order, i.e., can jump forward or backward by a number of elements.

The **`vector`** and **`deque`** containers support random-access iterators, and the **`list`**, **`set`**, **`multiset`**, **`map`**, and **`multimap`** containers support bidirectional iterators. Note that the **`stack`**, **`queue`**, and **`priority_queue`** don't support iterators, as shown in Table 22.4.

Table 22.4

Iterator Types Supported by Containers

STL Container	Type of Iterators Supported
vector	Random-access iterators
deque	Random-access iterators
list	Bidirectional iterators
set	Bidirectional iterators
multiset	Bidirectional iterators
map	Bidirectional iterators
multimap	Bidirectional iterators
stack	No iterator support
queue	No iterator support
priority_queue	No iterator support

22.3.2 Iterator Operators

You can manipulate an iterator using the overloaded operators to move its position, access the elements, and compare them. Table 22.5 shows the operators supported by iterators.

Table 22.5

Operators Supported by Iterators

Operator	Description
All iterators	
++p	Preincrement an iterator.
p++	Postincrement an iterator.
Input iterators	
*p	Dereference an iterator (used as rvalue).
p1 == p2	Evaluates true if p1 and p2 point to the same element.
p1 != p2	Evaluates true if p1 and p2 point to different elements.
Output iterators	
*p	Dereference an iterator (used as lvalue).
Bidirectionl iterators	
--p	Predecrement an iterator.
p--	Postdecrement an iterator.
Random-access iterators	
p += i	Increment iterator p by i positions.
p -= i	Decrement iterator p by i positions.
p + i	Returns an iterator ith position after p.
p - i	Returns an iterator ith position before p.
p1 < p2	Returns true if p1 is before p2.
p1 <= p2	Returns true if p1 is before or equal to p2.
p1 > p2	Returns true if p1 is after p2.
p1 >= p2	Returns true if p1 is after p2 or equal to p2.
p[i]	Returns the element at the position p offset by i.

All the iterators support the pre- and postincrement operators (**++p** and **p++**). The input iterators also support the dereference operator (*****) as an rvalue, equality checking operator (**==**), and inequality checking operator (**!=**). The output iterators also support the dereference operator (*****) as an lvalue. The forward iterators support all functions provided in the input and output iterators. The bidirectional iterators support the pre- and postdecrement operators in addition to all the functions in the forward iterators. The random-access iterators support all the operators listed in this table.

Listing 22.3 demonstrates how to use these operators on iterators.

Listing 22.3 IteratorOperatorDemo.cpp

```
#include <iostream>
```

```
 2  #include <vector>
 3  using namespace std;
 4
 5  int main()
 6  {
 7    vector<int> intVector;
 8    intVector.push_back(10);
 9    intVector.push_back(20);
10    intVector.push_back(30);
11    intVector.push_back(40);
12    intVector.push_back(50);
13    intVector.push_back(60);
14
15    vector<int>::iterator p1 = intVector.begin();
16    for (; p1 != intVector.end(); p1++)
17    {
18      cout << *p1 << " ";
19    }
20
21    cout << endl << *(--p1) << endl;
22    cout << *(p1 - 3) << endl;
23    cout << p1[-3] << endl;
24    *p1 = 1234;
25    cout << *p1 << endl;
26
27    return 0;
28  }
```

Sample output

```
      10 20 30 40 50 60
      60
      30
      30
      1234
```

The **vector** class contains random-access iterators. The program creates a vector (line 7), appends six elements into it (lines 8–13), and obtains an iterator **p1** in line 15. Since the **vector** class contains the random-access iterators, all the operators in Table 22.5 can be applied to **p1**.

22.3.3 Predefined Iterators

The STL containers use the **typedef** keyword to predefine synonyms for iterators. The predefined typedefs for iterators are **iterator**, **const_iterator**, **reverse_iterator**, and **const_reverse_iterator**. These iterators are defined in every first-class container for consistency. So, you can use them uniformly in your application programs.

For example,

```
vector<int>::iterator p1;
```

defines **p1** to be an iterator for the **vector<int>** container.

```
list<int>::iterator p2;
```

defines **p2** to be an iterator for the **list<int>** container.

> NOTE:
>
> Since **iterator** is a **typedef** defined inside a class such as **vector**, the *scope resolution* operator is needed to reference it.

The **typedef const_iterator** is the same as **typedef iterator**, except that you cannot modify elements through a **const_iterator**. A **const_iterator** is read-only. Listing 22.4 shows the differences between **iterator** and **const_iterator**.

Listing 22.4 ConstIteratorDemo.cpp

```
 1  #include <iostream>
 2  #include <vector>
 3  using namespace std;
 4
 5  int main()
 6  {
 7    vector<int> intVector;
 8    intVector.push_back(10);
 9
10    vector<int>::iterator p1 = intVector.begin();
11    vector<int>::const_iterator p2 = intVector.begin();
12
13    *p1 = 123; // OK
14    *p2 = 123; // Not allowed
15
16    cout << *p1 << endl;
17    cout << *p2 << endl;
18
19    return 0;
20  }
```

Sample output

```
Error line 14: cannot modify a const object
```

Since **p2** is a **const_iterator** (line 11), you cannot modify the element from **p2**.

You use reverse iterators to traverse containers in the reverse direction. Listing 22.5 demonstrates how to use **reverse_iterator**.

Listing 22.5 ReverseIteratorDemo.cpp

```
 1  #include <iostream>
 2  #include <vector>
 3  using namespace std;
 4
 5  int main()
 6  {
 7    vector<int> intVector;
 8    intVector.push_back(10);
 9    intVector.push_back(30);
10    intVector.push_back(20);
11
12    vector<int>::reverse_iterator p1 = intVector.rbegin();
13    for (; p1 != intVector.rend(); p1++)
14    {
15      cout << *p1 << " ";
16    }
17
18    return 0;
19  }
```

Sample output

```
     20 30 10
```

The program declares a **reverse_iterator p1** in line 12. The function **rbegin()** returns a **reverse_iterator** that refers to the last element in the container (line 12). The function **rend()** returns a **reverse_iterator** that refers to the next element after the first element in the reversed container (line 13).

The **typedef const_reverse_iterator** is the same as **typedef reverse_iterator**, except that you cannot modify elements through a **const_reverse_iterator**. A **const_reverse_iterator** is read-only.

*22.3.4 **istream_iterator** and **ostream_iterator***

Iterators are used for sequencing elements. You can use iterators to sequence the elements in a container as well as the elements in input/output streams. Listing 22.6 demonstrates how to use **istream_iterator** to input data from an input stream and **ostream_iterator** to output data to an output stream. The program prompts the user to enter three integers and displays the largest integer.

Listing 22.6 InputOutputStreamIteratorDemo.cpp

```
 1  #include <iostream>
 2  #include <iterator>
 3  #include <cmath>
 4  using namespace std;
 5
 6  int main()
 7  {
 8    cout << "Enter three numbers: ";
 9    istream_iterator<int> inputIterator(cin);
10    ostream_iterator<int> outputIterator(cout);
11
12    int number1 = *inputIterator;
13    inputIterator++;
14    int number2 = *inputIterator;
15    inputIterator++;
16    int number3 = *inputIterator;
17
18    cout << "The largest number is ";
19    *outputIterator = max(max(number1, number2), number3);
20
21    return 0;
22  }
```

Sample output

```
Enter three numbers: 34 12 23 ↵Enter
The largest number is 34
```

The **istream_iterator** and **ostream_iterator** are in the **<iterator>** header, so it is included in line 2. An **istream_iterator inputIterator** is created for reading integers from the **cin** object in line 9. An **ostream_iterator outputIterator** is created for writing integers to the **cout** object in line 10.

The dereferencing operator applies on **inputIterator** (line 12) to read an integer from **cin**, and the iterator is moved to point to the next number in the input stream (line 13).

The dereferencing operator applies on **outputIterator** (line 19) to write an integer to **cout**. Here ***outputIterator** is an lvalue.

This example demonstrates how to use the **istream_iterator** and **ostream_iterator**. Using **istream_iterator** and **ostream_iterator** seems to be a contrived way for console input and output. You will see the real benefits of using these iterators in the next chapter when we introduce STL algorithms.

Check Point

22.8 Does an iterator act like a pointer to an element? How do you obtain the iterator for the first element in a container? How do you obtain the iterator that points to the position after the last element in a container?

22.9 Show the output of the following code:

```
vector<int> intVector;
intVector.push_back(1);
intVector.push_back(2);
intVector.push_back(3);
intVector.push_back(4);

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

cout << "\nsize " << intVector.size() << " ";
```

22.10 List the types of iterators.

22.4 C+11 Auto Type Inference

Key Point: *The new C++11* **`auto`** *keyword can be used to declare a variable. The type of the variable is automatically determined by the compiler based on the type of the value assigned to the variable. This is known as auto type inference.*

In C++11, if the compiler is able to determine the type of a variable from its initialization, you can simply declare it using the **`auto`** keyword as shown in the following example:

```
auto x = 3;
auto y = x;
```

The compiler automatically determines that variable **`x`** is of the type **`int`** from the integer value **`3`** assigned to **`x`** and determines that variable **`y`** is of the type **`int`** since int variable **`x`** is assigned to **`y`**. This is not a good example of using the **`auto`** type. The real good use of the **`auto`** type is to replace long types. For example,

```
vector<int>::reverse_iterator p1 = intVector.rbegin();
```

is better replaced by

```
auto p1 = intVector.rbegin();
```

The new code is shorter and simpler and it relieves the programmer from writing a long and awkward type declaration.

22.5 Sequence Containers

Key Point: *A sequence container maintains the order for the elements in the container.*

The STL provides three sequence containers: **`vector`**, **`list`**, and **`deque`**. The **`vector`** and **`deque`** containers are implemented using arrays, and the **`list`** container is implemented using a linked list.

- A **`vector`** is efficient if the elements are appended to the **`vector`**, but it is expensive to insert or delete elements anywhere except at the end of the **`vector`**.
- A **`deque`** is like a vector, but it is efficient for insertion at both the front and end of a **`deque`**. Nevertheless, it is still expensive to insert or delete elements in the middle of a **`deque`**.
- A **`list`** is good for applications that require frequent insertion and deletion in the middle of a **`list`**.

A **`vector`** has the least overhead, a **`deque`** has slightly more overhead than a **`vector`**, and a **`list`** has the most overhead.

Tables 22.2 and 22.3 listed the functions common to all the containers and first-class containers. In addition to these common functions, each sequence container has the functions shown in Table 22.6.

Table 22.6

Common Functions in Sequence Containers

Functions	**Description**
assign(n, elem)	Assign n copies of the specified element in the container.
assign(beg, end)	Assign the elements in the range from iterator beg to iterator end.
push_back(elem)	Appends an element in the container.
pop_back()	Removes the last element from the container.
front()	Returns the first element in the container.
back()	Returns the last element in the container.
insert(position, elem)	Inserts an element at the specified iterator.

22.5.1 Sequence Container: ***`vector`***

As shown in Table 22.2, every container has a no-arg constructor, a copy constructor, and a destructor and supports the functions **`empty()`**, **`size()`**, and relational operators. Every first-class container contains the functions **`swap`**, **`max_size`**, **`clear`**, **`begin`**, **`end`**, **`rbegin`**, **`rend`**, and **`erase`**, as shown in Table 22.3, and the iterators, as shown in Table 22.5. Every sequence container contains the functions **`assign`**, **`push_back`**, **`pop_back`**, **`front`**, **`back`**, and **`insert`**, as shown in Table 22.6. Besides these common functions, the **`vector`** class also contains the functions shown in Table 22.7.

Table 22.7

Functions Specific in **`vector`**

Functions	**Description**
vector(n, element)	Constructs a vector filled with *n* copies of the same element.
vector(beg, end)	Constructs a vector initialized with elements from iterator beg to end.
vector(size)	Constructs a vector with the specified size.
at(index): dataType	Returns the element at the specified index.

Listing 22.7 demonstrates how to use the functions in **`vector`**.

Listing 22.7 VectorDemo.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  double values[] = {1, 2, 3, 4, 5, 6, 7};
  vector<double> doubleVector(values, values + 7);

  cout << "Initial contents in doubleVector: ";
  for (int i = 0; i < doubleVector.size(); i++)
    cout << doubleVector[i] << " ";

  doubleVector.assign(4, 11.5);

  cout << "\nAfter the assign function, doubleVector: ";
  for (int i = 0; i < doubleVector.size(); i++)
    cout << doubleVector[i] << " ";

  doubleVector.at(0) = 22.4;
  cout << "\nAfter the at function, doubleVector: ";
  for (int i = 0; i < doubleVector.size(); i++)
    cout << doubleVector[i] << " ";
24
25    auto itr = doubleVector.begin();
26    doubleVector.insert(itr + 1, 555);
27    doubleVector.insert(itr + 1, 666);
28    cout << "\nAfter the insert function, doubleVector: ";
29    for (int i = 0; i < doubleVector.size(); i++)
30      cout << doubleVector[i] << " ";
31
32    doubleVector.erase(itr + 2, itr + 4);
33    cout << "\nAfter the erase function, doubleVector: ";
34    for (int i = 0; i < doubleVector.size(); i++)
35      cout << doubleVector[i] << " ";
36
37    doubleVector.clear();
38    cout << "\Size is " << doubleVector.size() << endl;
39    cout << "Is empty? " <<
40          (doubleVector.empty() ? "true" : "false") << endl;
41
42    return 0;
43  }
```

Sample output

```
Initial contents in doubleVector: 1 2 3 4 5 6 7
After the assign function, doubleVector: 11.5 11.5 11.5 11.5
After the at function, doubleVector: 22.4 11.5 11.5 11.5
After the insert function, doubleVector: 22.4 666 555 11.5 11.5
11.5
After the erase function, doubleVector: 22.4 666 11.5 11.5
Size is 0
Is empty? true
```

The program creates an array of seven elements in line 7, and creates a vector using the elements from the array. Arrays can be accessed using pointers. The pointers are like iterators, so **`values`** points to the first element and **`values + 7`** to the position after last element in the array.

The program displays all the elements in the vector using a **`for`** loop (lines 17–18). The subscript operator **`[]`** (line 18) can be used for a **`vector`** or a **`deque`** to access elements in the container.

The program assigns **`22.4`** to the first element (line 20) in the vector using

```
doubleVector.at(0) = 22.4;
```

This statement is the same as

```
doubleVector[0] = 22.4;
```

Iterators can be used to specify the positions in a container. An iterator is obtained in line 25. A new element is inserted at position **`itr + 1`** (line 26), and another one is inserted in the same position **`itr + 1`** (line 27). The program deletes the elements from **`itr + 2`** to **`itr + 4 - 1`** (line 32).

22.5.2 Sequence Container: ***deque***

The term *deque* stands for *double-ended queue*. A **deque** provides efficient operations to support insertion and deletion on both its ends. Besides the common functions for all sequence containers, the **deque** class contains the functions shown in Table 22.8.

Table 22.8

Functions Specific in **deque**

Functions	**Description**
deque(n, element)	Constructs a deque filled with *n* copies of the same element.
deque(beg, end)	Constructs a deque initialized with elements from iterator beg to end.
deque(size)	Constructs a deque with the specified size.
at(index): dataType	Returns the element at the specified index.
push_front(element)	Inserts the element to the front of the queue.
pop_front(): dataType	Removes the element from the front of the queue.

Listing 22.8 demonstrates how to use the functions in **deque**.

Listing 22.8 DequeDemo.cpp

```
#include <iostream>
#include <deque>
using namespace std;

int main()
{
  double values[] = {1, 2, 3, 4, 5, 6, 7};
  deque<double> doubleDeque(values, values + 7);

  cout << "Initial contents in doubleDeque: ";
  for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

  doubleDeque.assign(4, 11.5);
  cout << "\nAfter the assign function, doubleDeque: ";
  for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

  doubleDeque.at(0) = 22.4;
  cout << "\nAfter the at function, doubleDeque: ";
  for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

  deque<double>::iterator itr = doubleDeque.begin();
25   doubleDeque.insert(itr + 1, 555);
26   doubleDeque.insert(itr + 1, 666);
27   cout << "\nAfter the insert function, doubleDeque: ";
28   for (int i = 0; i < doubleDeque.size(); i++)
29     cout << doubleDeque[i] << " ";
30
31   doubleDeque.erase(itr + 2, itr + 4);
32   cout << "\nAfter the erase function, doubleDeque: ";
33   for (int i = 0; i < doubleDeque.size(); i++)
34     cout << doubleDeque[i] << " ";
35
36   doubleDeque.clear();
37   cout << "\nAfter the clear function, doubleDeque: ";
38   cout << "Size is " << doubleDeque.size() << endl;
39   cout << "Is empty? " <<
40         (doubleDeque.empty() ? "true" : "false") << endl;
41
42   doubleDeque.push_front(10.10);
43   doubleDeque.push_front(11.15);
44   doubleDeque.push_front(12.34);
45   cout << "After the insertion, doubleDeque: ";
46   for (int i = 0; i < doubleDeque.size(); i++)
47     cout << doubleDeque[i] << " ";
48
49   doubleDeque.pop_front();
50   doubleDeque.pop_back();
51   cout << "\nAfter the pop functions, doubleDeque: ";
52   for (int i = 0; i < doubleDeque.size(); i++)
53     cout << doubleDeque[i] << " ";
54
55   return 0;
56 }
```

Sample output

```
Initial contents in doubleDeque: 1 2 3 4 5 6 7
After the assign function, doubleDeque: 11.5 11.5 11.5 11.5
After the at function, doubleDeque: 22.4 11.5 11.5 11.5
After the insert function, doubleDeque: 22.4 555 666 11.5 11.5
11.5
After the erase function, doubleDeque: 22.4 555 666 11.5
After the clear function, doubleDeque: Size is 0
Is empty? true
After the insertion, doubleDeque: 12.34 11.15 10.1
After the pop functions, doubleDeque: 11.15
```

The **`deque`** class contains all the functions in the **`vector`** class. So, you can use a **`deque`** wherever a **`vector`** is used. Lines 1–40 in Listing 22.8 are almost the same as lines 1–40 in Listing 22.7.

The **`push_front`** function is used to add elements to the front of the deque in lines 42–44, the **`pop_front()`** function removes the element from the front of the deque (line 49), and the **`pop_back()`** function removes the element from the back of the deque (line 50).

22.5.3 Sequence Container: ***list***

The class **list** is implemented as a doubly linked list. It supports efficient insertion and deletion operations anywhere on the list. Besides the common functions for all sequence containers, the **list** class contains the functions shown in Table 22.9.

Table 22.9

Functions Specific in **list**

Functions	**Description**
list(n, element)	Constructs a list filled with n copies of the same element.
list(beg, end)	Constructs a list initialized with elements from iterator beg to end-1.
list(size)	Constructs a list initialized with the specified size.
push_front(element)	Inserts the element to the front of the queue.
pop_front(): dataType	Removes the element from the front of the queue.
remove(element)	Removes all the elements that are equal to the specified element.
remove_if(oper)	Removes all the elements for which oper(element) is true.
splice(pos, list2)	All the elements of list2 are moved to this list before the specified position. After invoking this function, list2 is empty.
splice(pos1, list2, pos2)	All the elements of list2 starting from pos2 are moved to this list before pos1. After invoking this function, list2 is empty.
splice(pos1, list2, beg, end)	All the elements of list2 from iterator beg to end are moved to this list before pos1. After invoking this function, list2 is empty.
sort()	Sorts the elements in the list in increasing order.
sort(oper)	Sorts the elements in the list. The sort criterion is specified by oper.
merge(list2)	Suppose the elements in this list and list2 are sorted. Merges list2 into this list. After the merge, list2 is empty.
merge(list2, oper)	Suppose the elements in this list and list2 are sorted based on sort criterion oper. Merges list2 into this list.
reverse()	Reverse the elements in this list.

The iterators for **vector** and **deque** are random-access but are bidirectional for **list**. You cannot access the elements in a list using the subscript operator **[]**. Listing 22.9 demonstrates how to use the functions in **list**.

Listing 22.9 ListDemo.cpp

```
#include <iostream>
#include <list>
using namespace std;

int main()
```

```
{
  int values[] = {1, 2, 3, 4};
  list<int> intList(values, values + 4);

  cout << "Initial contents in intList: ";
  for (int& e: intList)
    cout << e << " ";

  intList.assign(4, 11);
  cout << "\nAfter the assign function, intList: ";
  for (int& e: intList)
    cout << e << " ";

  auto itr = intList.begin();
  itr++;
  intList.insert(itr, 555);
  intList.insert(itr, 666);
  cout << "\nAfter the insert function, intList: ";
  for (int& e: intList)
    cout << e << " ";

  auto beg = intList.begin();
  itr++;
  intList.erase(beg, itr);
  cout << "\nAfter the erase function, intList: ";
  for (int& e: intList)
    cout << e << " ";

  intList.clear();
  cout << "\nAfter the clear function, intList: ";
  cout << "Size is " << intList.size() << endl;
  cout << "Is empty? " <<
        (intList.empty() ? "true" : "false");

  intList.push_front(10);
  intList.push_front(11);
  intList.push_front(12);
  cout << "\nAfter the push functions, intList: ";
  for (int& e: intList)
    cout << e << " ";

  intList.pop_front();
  intList.pop_back();
  cout << "\nAfter the pop functions, intList: ";
  for (int& e: intList)
    cout << e << " ";

  int values1[] = {7, 3, 1, 2};
  list<int> list1(values1, values1 + 4);
  list1.sort();
  cout << "\nAfter the sort function, list1: ";
  for (int& e: list1)
    cout << e << " ";

  list<int> list2(list1);
  list1.merge(list2);
  cout << "\nAfter the merge function, list1: ";
  for (int& e: list1)
    cout << e << " ";
65    cout << "\nSize of list2 is " << list2.size();
66
67    list1.reverse();
68    cout << "\nAfter the reverse function, list1: ";
69    for (int& e: list1)
70      cout << e << " ";
71
72    list1.push_back(7);
73    list1.push_back(1);
74    cout << "\nAfter the push functions, list1: ";
75    for (int& e: list1)
76      cout << e << " ";
77
78    list1.remove(7);
79    cout << "\nAfter the remove function, list1: ";
80    for (int& e: list1)
81      cout << e << " ";
82
83    list2.assign(7, 2);
84    cout << "\nAfter the assign function, list2: ";
85    for (int& e: list2)
86      cout << e << " ";
87
88    auto p = list2.begin();
89    p++;
90    list2.splice(p, list1);
91    cout << "\nAfter the splice function, list2: ";
92    for (int& e: list2)
93      cout << e << " ";
94    cout << "\nAfter the splice function, list1's size is "
95      << list1.size();
96
97    return 0;
98  }
```

Sample output

```
Initial contents in intList: 1 2 3 4
After the assign function, intList: 11 11 11 11
After the insert function, intList: 11 555 666 11 11 11
After the erase function, intList: 11 11
After the clear function, intList: Size is 0
Is empty? true
After the push functions, intList: 12 11 10
After the pop functions, intList: 11
After the sort function, list1: 1 2 3 7
After the merge function, list1: 1 1 2 2 3 3 7 7
Size of list2 is 0
After the reverse function, list1: 7 7 3 3 2 2 1 1
After the push functions, list1: 7 7 3 3 2 2 1 1 7 1
After the remove function, list1: 3 3 2 2 1 1 1
After the assign function, list2: 2 2 2 2 2 2 2
After the splice function, list2: 2 3 3 2 2 1 1 1 2 2 2 2 2 2
After the splice function, list1's size is 0
```

The program creates a list **`intList`** and displays its contents in lines 7–13.

The program assigns four elements with value **`11`** to **`intList`** (line 15), inserts **`555`** and **`666`** into the position specified by the iteration **`itr`** (lines 22–23), erases the elements from the **`beg`** iterator to iterator **`itr`** (line 30), clears the list (line 35), and pushes and pops elements (lines 41–52).

The program creates a list **`list1`** (line 55), sorts it (line 56), and merges it with **`list2`** (line 62). After the merge, **`list2`** is empty.

The program reverses **`list1`** (line 68), and removes all the elements with value **`7`** from **`list1`** (line 79).

The program applies the **`splice`** function to move all the elements from **`list1`** into **`list2`** before the iterator **`p`** (line 91). Afterwards, **`list1`** is empty.

Check Point

22.11 For what applications should you use a **`vector`**, a **`deque`**, or a **`list`**? What types of the iterators are supported in **`vector`**, **`deque`**, and **`list`**?

22.12 What is wrong in the following code?

```
vector<int> intVector;
intVector.assign(4, 20);
intVector.insert(1, 10);
```

22.13 How do you remove elements in a **`vector`**, a **`deque`**, or a **`list`**?

22.14 Are the **`sort`**, **`splice`**, **`merge`**, and **`reverse`** functions contained in **`vector`**, **`deque`**, or **`list`**?

22.6 Associative Containers

Key Point: *The elements stored in an associative container can be accessed through keys.*

The STL provides four associative containers: **`set`**, **`multiset`**, **`map`**, and **`multimap`**. These containers provide fast storage and quick access to retrieve elements using *keys* (often called search *keys*). Elements in an associative container are sorted according to some sorting criterion. By default, the elements are sorted using the **`<`** operator.

A set stores a set of keys. A map stores a set of key/value pairs. The first element in the pair is the key and the second element is a value.

The **set** and **multiset** containers are identical except that a **multiset** allows duplicate keys and a **set** does not. The **map** and **multimap** are identical except that a **multimap** allows duplicate keys and a **map** does not.

Tables 22.2 and 22.3 listed the functions common to all the containers and first-class containers. Besides these common functions, each associative container supports those shown in Table 22.10.

Table 22.10

Common Functions in Associative Containers

Functions	**Description**
find(key)	Returns an iterator that points to the element with the specified key in the container.
lower_bound(key)	Returns an iterator that points to the first element with the specified key in the container.
upper_bound(key)	Returns an iterator that points to the next element after the last element with the specified key in the container.
count(key)	Returns the number of occurrences of the element with the specified key in the container.

*22.6.1 Associative Containers: **set** and **multiset***

The elements are the keys stored in a **set**/**multiset** container. A multiset allows duplicate keys, but a set does not. Listing 22.10 demonstrates how to use the **set** and **multiset** containers.

Listing 22.10 SetDemo.cpp

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
  int values[] = {3, 5, 1, 7, 2, 2};
  multiset<int> set1(values, values + 6);

  cout << "Initial contents in set1: ";
  for (int e: set1)
    cout << e << " ";

14    set1.insert(555);
15    set1.insert(1);
16    cout << "\nAfter the insert function, set1: ";
17    for (int e: set1)
18      cout << e << " ";
19
20    auto p = set1.lower_bound(2);
21    cout << "\nLower bound of 2 in set1: " << *p;
22    p = set1.upper_bound(2);
23    cout << "\nUpper bound of 2 in set1: " << *p;
24
25    p = set1.find(2);
26    if (p == set1.end())
27      cout << "2 is not in set1" << endl;
28    else
29      cout << "\nThe number of 2's in set1: " << set1.count(2);
30
31    set1.erase(2);
32    cout << "\nAfter the erase function, set1: ";
33    for (int e: set1)
34      cout << e << " ";
35
36    return 0;
37  }
```

Sample output

```
Initial contents in set1: 1 2 2 3 5 7
After the insert function, set1: 1 1 2 2 3 5 7 555
Lower bound of 2 in set1: 2
Upper bound of 2 in set1: 3
The number of 2's in set1: 2
After the erase function, set1: 1 1 3 5 7 555
```

The program creates a set **set1** and displays its contents in lines 7–13. By default, the elements in a set are sorted in increasing order. To specify a decreasing order, you may replace line 8 by

```
multiset<int, greater<int>> set1(values, values + 6);
```

The program inserts keys **555** and **1** (lines 15–16) and displays the new elements in the set (lines 17–19). Invoking **lower_bound(2)** (line 21) returns the iterator that points to the first occurrence of **2** in the container, and invoking **upper_bound(2)** (line 23) returns the iterator that points to the next element after the last occurrence of **2** in the container. Thus, ***p** in line 23 displays element **3**.

Invoking **find(2)** (line 26) returns the iterator that points to the first occurrence of **2** in the container. If no such element is in the container, the returned iterator is **end()** (line 27).

Invoking **erase(2)** (line 32) deletes from the set all the elements with key value **2**.

This example created a **multiset**. You can replace **multiset** by **set** as follows:

```
set<int> set1(values, values + 6);
```

Trace the program with this new statement.

*22.6.2 Associative Containers: **map** and **multimap***

Each element in a **map**/**multimap** is a pair. The first value in the pair is the key, and the second value is associated with the key. A **map**/**multimap** provides quick access to values using the key. A **multimap** allows duplicate keys, but a **map** does not. Listing 22.11 demonstrates how to use the **map** and **multimap** containers.

Listing 22.11 MapDemo.cpp

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<int, string> map1;
  map1.insert(map<int, string>::value_type(100, "John Smith"));
  map1.insert(map<int, string>::value_type(101, "Peter King"));
  map1.insert(map<int, string>::value_type(102, "Jane Smith"));
  map1.insert(map<int, string>::value_type(103, "Jeff Reed"));

  cout << "Initial contents in map1:\n";
  map<int, string>::iterator p;
  for (p = map1.begin(); p != map1.end(); p++)
    cout << p->first << " " << p->second << endl;

  cout << "Enter a key to serach for the name: ";
  int key;
  cin >> key;
  p = map1.find(key);

  if (p == map1.end())
    cout << "  Key " << key << " not found in map1";
  else
    cout << "  " << p->first << " " << p->second << endl;

  map1.erase(103);
  cout << "\nAfter the erase function, map1:\n";
  for (p = map1.begin(); p != map1.end(); p++)
    cout << p->first << " " << p->second << endl;

34    return 0;
35  }
```

Sample output

```
Initial contents in map1:
100 John Smith
101 Peter King
102 Jane Smith
103 Jeff Reed

Enter a key to serach for the name: 105 ↵Enter
  Key 105 not found in map1

After the erase function, map1:
100 John Smith
101 Peter King
102 Jane Smith
```

The program creates a map **`map1`** using its no-arg constructor (line 8), and inserts key/value pairs to **`map1`** (lines 9–12). You can insert a pair to a map using the **`insert`** function (lines 9-10) or using the syntax **`map1[key] = value`** (lines 11-12).

To insert a pair using the **`insert`** function, you have to create a pair using the **`value_type(key, value)`** function.

The **`map1[key] = value`** syntax is shorter and more readable than using the **`insert`** function. If the key is not in the map, the syntax inserts a new pair to the map. If the key is already in the map, the syntax replaces the existing pair with the new pair. Given a key, you can also use **`map1[key]`** to retrieve the value for the key.

The program prompts the user to enter a key (lines 19–21). Invoking **`find(key)`** returns the iterator that points to the element with the specified key (line 22). A pair consists of the key and the value, which can be accessed using **`p->first`** and **`p->second`** (line 27).
Invoking **`erase(103)`** deletes the element with key **`103`** (line 29).

This example created a **`map`**. You can replace **`map`** by **`multipmap`** as follows:

```
multimap<int, string> map1;
```

The program runs exactly the same as using a **map**.

Check Point

22.15 For what applications should you use a **set** or **multiset**? What are the differences between **set** and **multiset**?

22.16 Show the output of the following code:

```
set<int> intSet;
intSet.insert(20);
intSet.insert(10);
intSet.erase(30);
intSet.insert(10);

set<int>::iterator p;
for (p = intSet.begin(); p != intSet.end(); p++)
{
  cout << *p << " ";
}
```

22.17 Show the output of the following code:

```
multiset<int> intSet;
intSet.insert(20);
intSet.insert(10);
intSet.erase(30);
intSet.insert(10);

set<int>::iterator p;
for (p = intSet.begin(); p != intSet.end(); p++)
{
  cout << *p << " ";
}
```

22.18 What is wrong in the following code?

```
set<int> intSet;
intSet.insert(20);
intSet.insert(10);
cout << "\nfind 40? " << (intSet.find(40) ? "true" : "false");
```

22.19 For what applications should you use a **map** or **multimap**? What are the differences between **map** and **multimap**?

22.20 Show the output of the following code:

```
map<int, string> map1;
map1.insert(map<int, string>::value_type(100, "John Smith"));
map1.insert(map<int, string>::value_type(101, "Peter King"));
map1.insert(map<int, string>::value_type(100, "Jane Smith"));

map<int, string>::iterator p;
```

```
for (p = map1.begin(); p != map1.end(); p++)
{
  cout << p->first << " " << p->second << endl;
}
```

22.21 Show the output of the following code:

```
multimap<int, string> map1;
map1.insert(map<int, string>::value_type(100, "John Smith"));
map1.insert(map<int, string>::value_type(101, "Peter King"));
map1.insert(map<int, string>::value_type(100, "Jane Smith"));

map<int, string>::iterator p;
for (p = map1.begin(); p != map1.end(); p++)
{
  cout << p->first << " " << p->second << endl;
}
```

22.22 What is the header file for **`set`**? What is the header file for **`multiset`**? What is the header file for **`map`**? What is the header file for **`multimap`**?

22.7 Container Adapters

Key Point*: **`stack`**, **`queue`**, and **`priority_queue`** are called container adapters in STL.*

The STL provides three container adapters: **`stack`**, **`queue`**, and **`priority_queue`**. They are called *adapters* because they are adapted from the sequence containers for handling special cases. The STL enables the programmer to choose an appropriate sequence container for a container adapter. For example, you can create a stack with the underlying data structure **`vector`**, **`deque`**, or **`list`**.

Container adapters do not have iterators. Table 22.2 listed the functions common to all the containers. Besides these common functions, each container adapter supports the **`push`** and **`pop`** functions to insert and remove an element.

22.7.1 Container Adapter: **`stack`**

A **`stack`** is a last-in, first-out container. You can choose a **`vector`**, **`deque`**, or **`list`** to construct a **`stack`**. By default, a **`stack`** is implemented with a **`deque`**. The common functions on a **`stack`** are listed in Table 22.11.

Table 22.11

Functions in **stack**

Functions	Description
push(element)	Inserts the element to the top of the stack.
pop()	Removes an element from the top of the stack.
top()	Returns the top element from the stack without removing it.
size()	Returns the size of the stack.
empty()	Returns true if the stack is empty.

Listing 22.12 gives an example on how to use **stack**.

Listing 22.12 StackDemo.cpp

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

template<typename T>
void printStack(T &stack)
{
  while (!stack.empty())
  {
    cout << stack.top() << " ";
    stack.pop();
  }
}

int main()
{
  stack<int> stack1;
  stack<int, vector<int>> stack2;

  for (int i = 0; i < 8; i++)
  {
    stack1.push(i);
    stack2.push(i);
  }

  cout << "Contents in stack1: ";
  printStack(stack1);

  cout << "\nContents in stack2: ";
  printStack(stack2);

  return 0;
}
```

Sample output

```
      Contents in stack1: 7 6 5 4 3 2 1 0
      Contents in stack2: 7 6 5 4 3 2 1 0
```

This program creates a **stack** using the default implementation in line 18 and a **stack** using the **vector** implementation in line 23.

The program inserts numbers from **0** to **7** to **stack1** and **stack2** (lines 21–25) and invokes **printStack(stack1)** and **printStack(stack2)** to display and remove all the elements in **stack1** and **stack2**.

22.7.2 Container Adapter: ***queue***

A **queue** is a first-in, first-out container. You can choose a **deque** or **list** to construct a **queue**. By default, a **queue** is implemented with a **deque**. The common functions in a **queue** are listed in Table 22.12.

Table 22.12

Functions in **queue**

Functions	Description
push(element)	Inserts the element to the top of the queue.
pop()	Removes an element from the top of the queue.
front()	Returns the front element from the queue without removing it.
back()	Returns the back element from the queue without removing it.
size()	Returns the size of the queue.
empty()	Returns true if the queue is empty.

Listing 22.13 gives an example of how to use **queue**.

Listing 22.13 QueueDemo.cpp

```
#include <iostream>
#include <queue>
#include <list>
using namespace std;

template<typename T>
void printQueue(T &queue)
{
 9    while (!queue.empty())
10    {
11      cout << queue.front() << " ";
12      queue.pop();
13    }
14  }
15
16  int main()
17  {
18    queue<int> queue1;
19    queue<int, list<int> > queue2;
20
21    for (int i = 0; i < 8; i++)
22    {
23      queue1.push(i);
24      queue2.push(i);
25    }
26
27    cout << "Contents in queue1: ";
28    printQueue(queue1);
29
30    cout << "\nContents in queue2: ";
31    printQueue(queue2);
32
33    return 0;
34  }
```

Sample output

```
      Contents in queue1: 0 1 2 3 4 5 6 7
      Contents in queue2: 0 1 2 3 4 5 6 7
```

This program creates a **`queue`** using the default implementation in line 18 and a **`queue`** using the **`list`** implementation in line 22.

The program inserts numbers from **`0`** to **`7`** to **`queue1`** and **`queue2`** (lines 21–25), and invokes **`printQueue(queue1)`** and **`printQueue(queue2)`** to display and remove all the elements in **`queue1`** and **`queue2`**.

22.7.3 Container Adapter: ***`priority_queue`***

In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first.

You can choose a **vector** or **deque** to construct a **priority_queue**. By default, a **priority_queue** is implemented with a **vector**. For example, you may create a **priority queue** for **int** values using the following statements:

```
priority_queue<int> priority_queue1;
priority_queue<int, deque<int>> priority_queue2;
```

Be default, the elements are compared using the **<** operator. The largest value is assigned the highest priority. You can specify the **>** operator to construct a **priority queue** so that the smallest value is assigned the highest priority.

```
priority_queue<int, deque<int>, greater<int>> priority_queue3;
```

The **priority_queue** class uses the same functions **push**, **pop**, **top**, **size**, and **empty** as in the **stack** class.

Listing 22.14 gives an example on how to use **priority_queue**.

Listing 22.14 PriorityQueueDemo.cpp

```
#include <iostream>
#include "PriorityQueue.h"
#include <string>
using namespace std;

class Patient
{
public:
  Patient(const string& name, int priority)
  {
    this->name = name;
    this->priority = priority;
  }

  bool operator<(const Patient& secondPatient)
  {
    return (this->priority < secondPatient.priority);
  }

  bool operator>(const Patient& secondPatient)
  {
    return (this->priority > secondPatient.priority);
  }

  string getName()
  {
    return name;
  }

  int getPriority()
  {
    return priority;
  }

private:
  string name;
  int priority;
};

int main()
{
  // Queue of patients
  PriorityQueue<Patient> patientQueue;
  patientQueue.enqueue(Patient("John", 2));
  patientQueue.enqueue(Patient("Jim", 1));
  patientQueue.enqueue(Patient("Tim", 5));
  patientQueue.enqueue(Patient("Cindy", 7));

  while (patientQueue.getSize() > 0)
  {
    Patient element = patientQueue.dequeue();
    cout << element.getName() << " (priority: " <<
      element.getPriority() << ") ";
  }

  return 0;
}
```

Sample output

```
      Contents in queue1: 9 7 4 2 1
      Contents in queue2: 1 2 4 7 9
```

This program creates a **priority_queue** using the default implementation in line 18 and a **priority_queue** using the **deque** implementation with the **>** operator in line 19.

The program inserts numbers to **queue1** and **queue2** (lines 21–25) and invokes **printQueue(queue1)** and **printQueue(queue2)** to display and remove all the elements in **queue1** and **queue2**. In **queue1**, the largest value has the highest priority, but in **queue2**, the smallest number has the highest priority.

Check Point

22.23 Why container adapters are called adapters? Do container adapters have iterators?

22.24 Can you create a **stack**, **queue**, or **priority_queue** using a **vector**, **deque**, or **list**?

22.25 How do you insert elements to a `priority_queue`? How do you remove elements from a `priority_queue`? How do you find the size of a `priority_queue`?

Key Terms

- **associative container**
- **bidirectional iterator**
- **container**
- **container adapter**
- **deque**
- **first-class container**
- **forward iterator**
- **input iterator**
- **istream_iterator**
- **iterator**
- **list**
- **map**
- **multiset**
- **multimap**
- **ostream_iterator**
- **output iterator**
- **priority_queue**
- **random-access iterator**
- **queue**
- **sequence container**
- **set**
- **STL algorithm**
- **vector**

Chapter Summary

1. The *Standard Template Library* (*STL*) contains useful data structures. You can use them without having to reinvent the wheel.
2. A container object such as a vector is used to store a collection of data, often referred to as *elements*.
3. The STL container classes make extensive use of iterators, which are objects that facilitate traversing through the elements in a container. Iterators are like built-in pointers that provide a convenient way to access and manipulate the elements in a container.
4. The sequence containers (also known as sequential containers) represent linear data structures. The three sequence containers are **`vector`**, **`list`**, and **`deque`**.
5. Associative containers are nonlinear containers that can locate elements stored in the container quickly. Such containers can store sets of values or *key/value* pairs. The four associative containers are **`set`**, **`multiset`**, **`map`**, and **`multimap`**.
6. Container adapters are constrained versions of sequence containers. They are adapted from sequence containers for handling special cases. The three container adapters are **`stack`**, **`queue`**, and **`priority_queue`**.
7. An iterator is an abstraction of a pointer, and in fact, it is typically implemented using a pointer. Each container has its own iterator type. The abstraction hides the detailed implementation and provides a uniform way for using iterators on all containers.
8. Iterators can be classified into five categories: input iterators, output iterators, forward iterators, bidirectional iterators, and random-access iterators.
9. An input iterator is used for reading an element from a container.
10. An output iterator is used for writing an element to a container.
11. A forward iterator combines all the functionalities of input and output iterators to support both read and write operations.
12. A bidirectional iterator is a forward iterator with the capability of moving backward.
13. A random-access iterator is a bidirectional iterator with the capability of accessing any element in any order.

14. The iterator type determines which operators can be used. The **vector** and **deque** containers support random-access iterators, and the **list**, **set**, **multiset**, **map**, and **multimap** containers support bidirectional iterators. The **stack**, **queue**, and **priority_queue** don't support iterators.

15. A vector is efficient if the elements are appended to the vector. It is expensive to insert or delete elements in the middle of a vector.

16. A deque is like a vector, but it is efficient for insertion at both the front and end of a deque. It is still expensive to insert or delete elements in the middle of a deque.

17. A linked list is good for applications that require frequent insertion and deletion in the middle of a list.

18. The **set** and **multiset** containers are identical, except that a **multiset** allows duplicate keys and a **set** does not.

19. The **map** and **multimap** are identical, except that a **multimap** allows duplicate keys and a **map** does not.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

22.1* (*Maximum and minimum*) Implement the following functions that find the maximum and minimum elements in a first-class container:

```
template<typename ElementType, typename ContainerType>
ElementType maxElement(ContainerType& container)

template<typename ElementType, typename ContainerType>
ElementType minElement(ContainerType& container)
```

22.2* (*Position of a value*) Implement the following function that finds the position of a specified value in a first-class container. Return **-1** if there is no match.

```
template<typename ElementType, typename ContainerType>
ElementType find(ContainerType& container,
  const ElementType& value)
```

22.3* (*Occurrence of a value*) Implement the following function that finds the number of occurrences of a specified value in a first-class container:

```
template<typename ElementType, typename ContainerType>
int countElement(ContainerType& container, const ElementType& value)
```

22.4* (*Reverse a container*) Implement the following function that reverses the elements in a container:

```
template<typename ContainerType>
void reverse(ContainerType& container)
```

22.5* (*Remove elements*) Implement the following function that removes the specified value from a first-class container. Only the first occurrence of a matching value in the container is removed.

```
template<typename ElementType, typename ContainerType>
void remove(ContainerType& container, const ElementType& value)
```

22.6* (*Replace elements*) Implement the following function that replaces a given element with a new value:

```
template<typename ElementType, typename ContainerType>
void replace(ContainerType& container,
  const ElementType& oldValue, const ElementType& newValue)
```

22.7** (*Union of two* **`sets`**) Implement the following mathematical set union function to combine two sets **`s1`** and **`s2`** into a new set **`s3`**:

```
template<typename ElementType>
void setUnion(set<ElementType>& s1,
  set<ElementType>& s2, set<ElementType>& s3)
```

22.8** (*Difference of two* **`sets`**) Implement the following mathematical set difference function to produce a new set **`s3`** from the difference between **`s1`** and **`s2`**:

```
template<typename ElementType>
void difference(set<ElementType>& s1,
  set<ElementType>& s2, set<ElementType>& s3)
```

22.9** (*Display nonduplicate words in ascending order*) Write a program that reads words from a text file and displays all the nonduplicate words in ascending order. (*Hint*: Use a set to store all the words.)

22.10** (*Display duplicate words in ascending order*) Write a program that reads words from a text file and displays all the words (duplicates allowed) in ascending order. (*Hint*: Use a multiset to store all the words.)

22.11** (*Count the keywords in C++ source code*) Write a program that reads a C++ source-code file and reports the number of keywords in the file. (*Hint*: Create a set to store all the C++ keywords.)

22.12** (*Count the occurrences of numbers entered*) Write a program that reads an unspecified number of integers and finds the occurrences of each number. Your input ends when the input is **0**.

Here is a sample run of the program:

Sample output

```
    Enter numbers (ending with 0): 2 3 4 1 2 34 4 3 0
    number of occurrences for 1 is 1
    number of occurrences for 2 is 2
    number of occurrences for 3 is 2
    number of occurrences for 4 is 2
    number of occurrences for 34 is 1
```

(*Hint*: Use a map to store pairs. The first element in the pair is a number entered from the input, and the second element tracks the number of occurrences of this number.)

22.13**(*Count the occurrences of words*) Write a program that prompts the user to enter a file name, counts the occurrences of words in the file, and displays the words and their occurrences in ascending order of words. Words are separated by spaces. The program uses a map to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value **1** to the map. Otherwise, increase the value for the word (key) by **1** in the map. Here is a sample run of the program:

Sample output

```
    Enter a file name: c:\test.txt ↵ Enter
    number of occurrences for bad is 2
    number of occurrences for good is 3
    number of occurrences for goodbye is 31
    number of occurrences for green is 3
    number of occurrences for red is 2
    number of occurrences for yellow is 12
```

22.14** (*Guess the capitals using maps*) Rewrite Programming Exercise 10.9 to store pairs of state and its capital in a map. Your program should prompt the user to enter a state and displays the capital for the state.

22.15** (*Count the occurrence of each keyword in C++ source file*) Write a program that reads a C++ source-code file and reports the occurrence of each keyword in the file. Here is a sample run:

Sample output

```
    Enter a C++ source file name: Welcome.cpp  ↵ Enter
    int occurs 3 times
    void occurs 1 time
    ...
    static occurs 1 time
```

22.16* (*Count consonants and vowels*) Write a program that prompts the user to enter a text file name and displays the number of vowels and consonants in the file. Use a set to store the vowels **A**, **E**, **I**, **O**, and **U**.

22.17*

(*Subtraction quiz*) Rewrite Programming Exercise 12.34 to store the answers in a set rather than a vector.