CHAPTER

19

STL Containers

Objectives

- To know the relationships among containers, iterators, and algorithms (§19.2).

- To distinguish sequence containers, associative containers, and container adapters (§19.2).

- To distinguish containers vector, deque, list, set, multiset, map, multimap, stack, queue, and priority queue (§19.2).

- To use common features of containers (§19.2).

- To access elements in a container using iterators (§19.3).

- To distinguish iterator types: input, output, forward, bidirectional, and random-access (§19.3.1).

- To manipulate iterators using operators (§19.3.2).

- To obtain iterators from containers and know the type of iterators supported by containers (§19.3.3).

- To perform input and output using istream_iterator and ostream_iterator (§19.3.4).

- To store, retrieve and process elements in sequence containers: vector, deque, list (§19.4).

- To store, retrieve and process elements in associative containers: set, multiset, map, and multimap (§19.5).

- To store, retrieve and process elements in container adapters: stack, queue, and priority queue (§19.6).
19.1 Introduction

Chapter 16, “Linked Lists, Stacks, and Queues,” and Chapter 17, “Trees, Heaps, and Priority Queues,” introduced several data structures such as linked lists, stacks, queues, heaps, and priority queues. These popular data structures are widely used 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 in the STL that you have learned is the vector class, which was introduced in §11.12. This chapter introduces the STL and you will learn how to use the classes in the STL to simplify application development.

19.2 STL Basics

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and was based on their research of generic programming in collaboration with David Musser. It is a collection of libraries written in C++. As its name suggests, 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. There are about 80 algorithms implemented in the STL. Most of these algorithms use iterators to access the elements in the container.
The STL containers can be classified into three categories:

<side remark: sequence containers>

- **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 non-linear 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**.

- **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**.

<side remark: three components>

Table 19.1 summarizes the container classes and their header files.

**Table 19.1**

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

<side remark: common functions>

<side remark: first-class container>

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 19.2 lists the common functions for all containers and Table 19.3 lists the common functions for the sequence containers and associative
containers. These two containers are also known as the first-class containers.

Table 19.2
Common Functions to All Containers

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

Table 19.3
Common Functions to First-Class Containers

<table>
<thead>
<tr>
<th>STL Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct1.swap(ct2)</td>
<td>Swaps the elements of two containers ct1 and ct2.</td>
</tr>
<tr>
<td>ct.max_size()</td>
<td>Returns the maximum number of elements a container can hold.</td>
</tr>
<tr>
<td>ct.clear()</td>
<td>Erases all elements from the container.</td>
</tr>
<tr>
<td>ct.begin()</td>
<td>Returns an iterator to the first element in the container.</td>
</tr>
<tr>
<td>ct.end()</td>
<td>Returns an iterator that refers to the next position after the end of the container.</td>
</tr>
<tr>
<td>ct.rbegin()</td>
<td>Returns an iterator to the last element in the container for processing elements in reverse order.</td>
</tr>
<tr>
<td>ct.rend()</td>
<td>Returns an iterator that refers to the position before the first element in the container.</td>
</tr>
<tr>
<td>ct.erase(beg, end)</td>
<td>Erases the elements in the container from beg to end-1. Both beg and end are iterators.</td>
</tr>
</tbody>
</table>

Listing 19.1 gives a simple example that demonstrates how to create a vector, list, deque, set, multiset, stack, and queue.

Listing 19.1 SimpleSTLDemo.cpp (Using STL Containers)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, A.***
#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#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: " << vector2.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;

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;

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;
deque1.swap(deque2);
cout << "size of deque1: " << deque1.size() << endl;
cout << "size of deque2: " << deque2.size() << endl;
cout << "deque1 < deque2? " << (deque1 < deque2) << endl << 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;

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;

<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

668
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

<end of output>

<Side Remark: no-arg constructor>
Each container has a non-arg constructor. The program creates vectors, lists, deques, sets, multisets, stacks, and queues in lines 12-18 using the container’s no-arg constructors.

<Side Remark: add element>
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, the push(element) function to pushes an element into a stack and queue in lines 92-95 and 101-104.

<Side Remark: set vs. multiset>
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.

<Side Remark: relational operator>
All containers support the relational operators. The program compares two containers of the same type in lines 31, 45, 58, and 72.

19.3 Iterators

Iterators are used extensively in the first-class containers for accessing and manipulating the elements. As you already have seen in Table 19.3 that several functions (e.g., begin() and end()) in the first-class containers are related to iterators.

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.

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 it to all other containers.

Listing 19.2 demonstrates using iterators in a vector and a set.

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

int main()
{
    vector<int> intVector;
    intVector.push_back(10);
    intVector.push_back(40);
    intVector.push_back(50);
    intVector.push_back(20);
    intVector.push_back(30);

    vector<int>::iterator p1;
    cout << "Traverse the vector: ";
    for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
    {
        cout << *p1 << " ";
    }

    set<int> intSet;
    intSet.insert(10);
    intSet.insert(40);
    intSet.insert(50);
    cout << "Traverse the set: ";
    for (auto i = intSet.begin(); i != intSet.end(); i++)
    {
        cout << *i << " ";
    }
}
```

670
intSet.insert(20);
intSet.insert(30);

set<int>::iterator p2;
cout << "\nTraverse the set: ";
for (p2 = intSet.begin(); p2 != intSet.end(); p2++)
{
    cout << *p2 << " ";
}
cout << endl;
return 0;

<output>
Traverse the vector: 10 40 50 20 30
Traverse the set: 10 20 30 40 50
<end of output>

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

An iterator p1 is declared in line 15:

vector<int>::iterator p1;

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

The expression (line 17)

<Side Remark: begin()>
    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)

<Side Remark: end()>
    p1 != intVector.end();

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

The expression (line 17)

<Side Remark: move iterator>
    p1++
advances the iterator to the next element.

The expression (line 19)

```cpp
*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.

19.3.1 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 operation.

- **Bidirectional iterator**: A bidirectional iterator is a forward iterator with the capability of moving backward. The iterator can be moved freely back or forth one at a time.

- **Random access iterator**: A random access iterator is a bidirectional iterator with the capability of accessing
any element in any order, i.e., to jump forward or backward by a number of elements.

<side remark: supported types> 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 19.4.

Table 19.4
Iterator Types Supported by Containers

<table>
<thead>
<tr>
<th>STL Container</th>
<th>Type of Iterators Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>random access iterators</td>
</tr>
<tr>
<td>deque</td>
<td>random access iterators</td>
</tr>
<tr>
<td>list</td>
<td>bidirectional iterators</td>
</tr>
<tr>
<td>set</td>
<td>bidirectional iterators</td>
</tr>
<tr>
<td>multiset</td>
<td>bidirectional iterators</td>
</tr>
<tr>
<td>map</td>
<td>bidirectional iterators</td>
</tr>
<tr>
<td>multimap</td>
<td>bidirectional iterators</td>
</tr>
<tr>
<td>stack</td>
<td>no iterator support</td>
</tr>
<tr>
<td>queue</td>
<td>no iterator support</td>
</tr>
<tr>
<td>priority_queue</td>
<td>no iterator support</td>
</tr>
</tbody>
</table>

19.3.2 Iterator Operators

You can manipulate the iterators using the overloaded operators to move its position, access the element, and compare them. Table 2.5 shows the operators supported by the iterators.

Table 19.5
Operators Supported by Iterators
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All iterators</td>
<td>All iterators provide basic operation.</td>
</tr>
<tr>
<td>++p</td>
<td>Preincrement an iterator.</td>
</tr>
<tr>
<td>p++</td>
<td>Postincrement an iterator.</td>
</tr>
<tr>
<td>Input iterators</td>
<td>All input iterators support dereference operator.</td>
</tr>
<tr>
<td>*p</td>
<td>Dereference an iterator (used as rvalue).</td>
</tr>
<tr>
<td>p1 == p2</td>
<td>Evaluates true if p1 and p2 point to the same element.</td>
</tr>
<tr>
<td>p1 != p2</td>
<td>Evaluates false if p1 and p2 point to the same element.</td>
</tr>
<tr>
<td>Output iterators</td>
<td>All output iterators support dereference operator.</td>
</tr>
<tr>
<td>*p</td>
<td>Dereference an iterator (used as lvalue).</td>
</tr>
<tr>
<td>Bidirectional iterators</td>
<td>All bidirectional iterators support dereference operator.</td>
</tr>
<tr>
<td>--p</td>
<td>Predecrement an iterator.</td>
</tr>
<tr>
<td>p--</td>
<td>Postdecrement an iterator.</td>
</tr>
<tr>
<td>Random-access iterators</td>
<td>All random-access iterators support dereference operator.</td>
</tr>
<tr>
<td>p += i</td>
<td>Increment iterator p by i positions.</td>
</tr>
<tr>
<td>p -= i</td>
<td>Decrement iterator p by i positions.</td>
</tr>
<tr>
<td>p + i</td>
<td>Returns an iterator ith position after p.</td>
</tr>
<tr>
<td>p - i</td>
<td>Returns an iterator ith position before p.</td>
</tr>
<tr>
<td>p1 &lt; p2</td>
<td>Returns true if p1 is before p2.</td>
</tr>
<tr>
<td>p1 &lt;= p2</td>
<td>Returns true if p1 is before or equal to p2.</td>
</tr>
<tr>
<td>p1 &gt; p2</td>
<td>Returns true if p1 is after p2.</td>
</tr>
<tr>
<td>p1 &gt;= p2</td>
<td>Returns true if p1 is after or equal to p2.</td>
</tr>
<tr>
<td>p[i]</td>
<td>Returns the element at the position p offset by i.</td>
</tr>
</tbody>
</table>

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

Listing 19.3 demonstrates how to use these operators on iterators.

Listing 19.3 IteratorOperatorDemo.cpp (Using Iterator Operators)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.

<Side Remark line 7: create vector>
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> intVector;
    intVector.push_back(10);
    intVector.push_back(20);
    intVector.push_back(30);
    intVector.push_back(40);
    intVector.push_back(50);
    intVector.push_back(60);

    vector<int>::iterator p1 = intVector.begin();
    for (; p1 != intVector.end(); p1++)
    {
        cout << *p1 << " ";
    }
    cout << endl << *(--p1) << endl;
    cout << *(p1 - 3) << endl;
    cout << *p1[-3] << endl;
    *p1 = 1234;
    cout << *p1 << endl;

    return 0;
}
```

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 15.5 can be applied to `p1`.

19.3.3 Predefined Iterators

The STL containers use the `typedef` keyword to predefine iterators. The `typedef` keyword is used to declare synonyms
for data types. For example, the following statement declares integer as a synonym for int.

    typedef int integer;

So now you can declarer a variable using

    integer value = 40;

The predefined typedefs are iterator, const iterator, reverse iterator, and const reverse iterator. These iterators are defined in every first-class container.

For example,

    <side remark: typedef iterator>
    vector<int>::iterator p1;

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

NOTE:

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

    <side remark: typedef const iterator>
    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 19.4 shows the differences between iterator and const iterator.

Listing 19.4 ConstIteratorDemo.cpp (Using Const Iterators)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

    <Side Remark line 10: iterator>
    <Side Remark line 11: const iterator>
    <Side Remark line 13: writable>
    <Side Remark line 14: read-only>
    <Side Remark line 16: read from p1>
    <Side Remark line 17: read from p2>

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

    int main()
    {   
        vector<int> intVector;
            intVector.push_back(10);
vector<int>::iterator p1 = intVector.begin();
vector<int>::const_iterator p2 = intVector.begin();

*p1 = 123; // OK
*p2 = 123; // Not allowed

cout << "p1 << endl;
cout << "p2 << endl;

return 0;

<output>
123
123
</output>

<end of output>

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

<side remark: typedef reverse_iterator>
You use reverse iterators to traverse containers in the reverse direction. Listing 19.5 demonstrates how to use reverse_iterator.

Listing 19.5 ReverseIteratorDemo.cpp (Using Reverse Iterators)
***PD: Please add line numbers (including space lines) in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.
<Side Remark line 7: create vector>
<Side Remark line 8: append element>
<Side Remark line 12: reverse_iterator>
<Side Remark line 13: rend()>
<Side Remark line 15: access element>

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

int main()
{  
  vector<int> intVector;
  intVector.push_back(10);
  intVector.push_back(30);
  intVector.push_back(20);

  vector<int>::reverse_iterator p1 = intVector.rbegin();
  for (; p1 != intVector.rend(); p1++)
  {  
    cout << "p1 << " << *p1 << " ";
  }
The program declares a reverse iterator \( p1 \) in line 12. The function \( \text{rbegin()} \) returns a reverse iterator that refers to the last element in the container (line 12). The function \( \text{rend()} \) returns a reverse iterator that refers to the next element after the first element in the reversed container (line 13).

**side remark: typedef reverse const_iterator**

The typedef \( \text{const reverse iterator} \) is the same as typedef reverse iterator except that you cannot modify elements through a \( \text{const reverse iterator} \). A \( \text{const reverse iterator} \) is read-only.

**19.3.4 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 19.6 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 19.6 InputOutputStreamIteratorDemo.cpp (Using I/O Iterators)

```cpp
#include <iostream>
#include <iterator>
#include <cmath>
using namespace std;

int main()
{
    cout << "Enter three numbers: ";
    istream_iterator<int> inputIterator(cin);
    ostream_iterator<int> outputIterator(cout);
    int number1 = *inputIterator;
```
```cpp
inputIterator++;  // Increment iterator
int number2 = *inputIterator;
inputIterator++;  // Increment iterator
int number3 = *inputIterator;

cout << "The largest number is ";
*outputIterator = max(max(number1, number2), number3);

return 0;
```

<output>
Enter three numbers: 34 12 23
The largest number is 34
<end of output>

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 a lvalue.

19.4 Sequence Containers

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 front and end of a deque, but 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 19.2 and 19.3 listed the functions common to all the containers and first-class containers. In addition to these
common functions, each sequence container has the following functions, as shown in Table 19.6.

Table 19.6

Common Functions in Sequence Containers

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign(n, elem)</td>
<td>Assign n copies of the specified element in the container.</td>
</tr>
<tr>
<td>assign(beg, end)</td>
<td>Assign the elements in the range from iterator beg to iterator end.</td>
</tr>
<tr>
<td>push_back(elem)</td>
<td>Appends an element in the container.</td>
</tr>
<tr>
<td>pop_back()</td>
<td>Removes the last element from the container.</td>
</tr>
<tr>
<td>front()</td>
<td>Returns the reference of the first element.</td>
</tr>
<tr>
<td>back()</td>
<td>Returns the reference of the last element.</td>
</tr>
<tr>
<td>insert(position, elem)</td>
<td>Inserts an element at the specified iterator.</td>
</tr>
</tbody>
</table>

19.4.1 Sequence Containers: vector

As shown in Table 19.2, every container has a non-arg constructor, a copy constructor, and 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 19.3, and the iterators, as shown in Table 19.5. Every sequence container contains the functions assign, push_back, pop_back, front, back, and insert, as shown in Table 19.6. In addition to these common functions, the vector class also contains the following functions, as shown in Table 19.7.

Table 19.7

Functions Specific in vector

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector(n, element)</td>
<td>Constructs a vector filled with n copies of the same element.</td>
</tr>
<tr>
<td>vector(beg, end)</td>
<td>Constructs a vector initialized with elements from iterator beg to end.</td>
</tr>
<tr>
<td>vector(size)</td>
<td>Constructs a vector with the specified size.</td>
</tr>
<tr>
<td>at(index): dataType</td>
<td>Returns the element at the specified index.</td>
</tr>
</tbody>
</table>

Listing 19.7 demonstrates how to use the functions in vector.

Listing 19.7 VectorDemo.cpp (Using Vectors)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

<Side Remark line 2: include vector>
```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 << "
After the assign function, doubleVector: ";
    for (int i = 0; i < doubleVector.size(); i++)
        cout << doubleVector[i] << " ";
    doubleVector.at(0) = 22.4;
    cout << "
After the at function, doubleVector: ";
    for (int i = 0; i < doubleVector.size(); i++)
        cout << doubleVector[i] << " ";
    vector<double>::iterator itr = doubleVector.begin();
    doubleVector.insert(itr + 1, 555);
    doubleVector.insert(itr + 1, 666);
    cout << "
After the insert function, doubleVector: ";
    for (int i = 0; i < doubleVector.size(); i++)
        cout << doubleVector[i] << " ";
    doubleVector.erase(itr + 2, itr + 4);
    cout << "
After the erase function, doubleVector: ";
    for (int i = 0; i < doubleVector.size(); i++)
        cout << doubleVector[i] << " ";
    doubleVector.clear();
    cout << "\nSize is " << doubleVector.size() << endl;
    cout << "Is empty? " << (doubleVector.empty() ? "true" : "false") << endl;
}
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 and values + 7 point to the first element and the last element in the array.

The program displays all the elements in the vector using a for loop (lines 16-17). The subscript operator [] (line 17) can be used for a vector or a deque to access elements in the container.

The program assigns 22.4 to the first element 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 26. A new element is inserted at position itr + 1 (line 27) and another one is inserted in the same position itr + 1 (line 28). The program deletes the elements from itr + 2 to itr + 4 (line 33).

19.4.2 Sequence Containers: deque

The term deque stands for double-ended queue. A deque provides efficient operations to support insertion and deletion on both ends of a deque. In addition to the common functions for all sequence containers, the deque class contains the following functions, as shown in Table 19.8.

Table 19.8

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 19.8 demonstrates how to use the functions in `deque`.

```
#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();
    doubleDeque.insert(itr + 1, 555);
    doubleDeque.insert(itr + 1, 666);
    cout << "\nAfter the insert function, doubleDeque: ";
    for (int i = 0; i < doubleDeque.size(); i++)
        cout << doubleDeque[i] << " ";
    doubleDeque.erase(itr + 2, itr + 4);
    cout << "\nAfter the erase function, doubleDeque: ";
```

Listing 19.8 DequeDemo.cpp (Using Deques)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

.Side Remark line 2: include deque
.Side Remark line 7: create an array
.Side Remark line 8: create deque
.Side Remark line 43: push front
.Side Remark line 49: pop front
for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

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

doubleDeque.push_front(10.10);
doubleDeque.push_front(11.15);
doubleDeque.push_front(12.34);
cout << "\nAfter the insertion, doubleDeque: ";
for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

doubleDeque.pop_front();
doubleDeque.pop_back();
cout << "\nAfter the pop functions, doubleDeque: ";
for (int i = 0; i < doubleDeque.size(); i++)
    cout << doubleDeque[i] << " ";

return 0;

<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 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
</end of output>

The deque class contains all the functions in the vector class. So if you can use a deque wherever a vector is used. Lines 1-42 in Listing 19.8 are almost the same as lines 1-42 in Listing 19.7.

The push front function is used to add elements to the front of the deque in lines 43-45, the pop front function removes the element from the front of the deque (line 50), and the pop back function removes the element from the back of the deque (line 51).

19.4.3 Sequence Containers: list

The class list is implemented as a doubly-linked list. It supports efficient insertion and deletion operations anywhere on the list. In addition to the common functions for all sequence containers, the list class contains the following functions, as shown in Table 19.9.

Table 19.9
## Functions Specific in `list`

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

The iterators for vector and deque are random-access, but is bidirectional for `list`. You cannot access the elements in a list using the subscript operator `[]`. Listing 19.9 demonstrates how to use the functions in `list`.

### Listing 19.9 ListDemo.cpp (Using Lists)

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

---

685
int main()
{
    int values[] = {1, 2, 3, 4};
    list<int> intList(values, values + 4);
    cout << "Initial contents in intList: ";
    list<int>::iterator p;
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    intList.assign(4, 11);
    cout << "After the assign function, intList: ";
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    list<int>::iterator itr = intList.begin();
    intList.insert(itr, 555);
    intList.insert(itr, 666);
    cout << "After the insert function, intList: ";
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    list<int>::iterator beg = intList.begin();
    intList.erase(beg, itr);
    cout << "After the erase function, intList: ";
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    intList.clear();
    cout << "After the clear function, intList: ";
    cout << "Size is " << intList.size() << endl;
    cout << "Is empty? " << (intList.empty() ? "true" : "false") << endl;
    intList.push_front(10);
    intList.push_front(11);
    intList.push_front(12);
    cout << "After the push functions, intList: ";
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    intList.pop_front();
    intList.pop_back();
    cout << "After the pop functions, intList: ";
    for (p = intList.begin(); p != intList.end(); p++)
        cout << "p = " << *p << " ";
    int values1[] = {7, 3, 1, 2};
    list<int> list1(values1, values1 + 4);
    list1.sort();
    cout << "After the sort function, list1: ";
    for (p = list1.begin(); p != list1.end(); p++)
        cout << "p = " << *p << " ";
cout << *p << " ";

list<int> list2(list1);
list1.merge(list2);
cout << "\nAfter the merge function, list1: ";
for (p = list1.begin(); p != list1.end(); p++)
cout << *p << " ";
cout << "\nSize of list2 is " << list2.size();

list1.reverse();
cout << "\nAfter the reverse function, list1: ";
for (p = list1.begin(); p != list1.end(); p++)
cout << *p << " ";

list1.push_back(7);
list1.push_back(1);
cout << "\nAfter the push functions, list1: ";
for (p = list1.begin(); p != list1.end(); p++)
cout << *p << " ";

list1.remove(7);
cout << "\nAfter the remove function, list1: ";
for (p = list1.begin(); p != list1.end(); p++)
cout << *p << " ";

list2.assign(7, 2);
cout << "\nAfter the assign function, list2: ";
for (p = list2.begin(); p != list2.end(); p++)
cout << *p << " ";
p = list2.begin();
p++;
list2.splice(p, list1);
cout << "\nAfter the splice function, list2: ";
for (p = list2.begin(); p != list2.end(); p++)
cout << *p << " ";
cout << "\nAfter the splice function, list1's size is " << list1.size();

return 0;
}

<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 2 3 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 1 7 1
After the remove function, list1: 3 3 2 2 1 1 1
After the assign function, list1: 2 2 2 2 2 2 2
After the splice function, list1: 2 3 3 3 2 2 1 1 1 2 2 2 2 2
After the splice function, list1’s size is 0
<end of output>
The program creates a list intList and displays its contents in lines 7-13.

The program assigns 4 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 it (line 35), and push and pop 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.

19.5 Associative Containers

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.

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 19.2 and 19.3 listed the functions common to all the containers and first-class containers. In addition to these common functions, each associative container supports the following functions, as shown in Table 19.10.

Table 19.10

Common Functions in Associative Containers
<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(key)</td>
<td>Returns an iterator that points to the element with the specified key in the container.</td>
</tr>
<tr>
<td>lower_bound(key)</td>
<td>Returns an iterator that points to the first element with the specified key in the container.</td>
</tr>
<tr>
<td>upper_bound(key)</td>
<td>Returns an iterator that points to the next element after the last element with the specified key in the container.</td>
</tr>
<tr>
<td>count(key)</td>
<td>Returns the number of occurrences of the element with the specified key in the container.</td>
</tr>
</tbody>
</table>

19.5.1 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 19.10 demonstrates how to use the `set` and `multiset` containers.

Listing 19.10 SetDemo.cpp (Using `set/multiset`)

```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: ", set1;
    set1.insert(555);
    set1.insert(1);
    multiset<int>::iterator p;
    for (p = set1.begin(); p != set1.end(); p++)
        cout << "*p << " ";
    set1.erase(p);
    cout << "
";
    return 0;
}
```

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

<Side Remark line 2: include list>
<Side Remark line 7: create an array>
<Side Remark line 8: create set1>
<Side Remark line 11: set1 iterator>
<Side Remark line 13: access element>
<Side Remark line 15: insert element>
<Side Remark line 21: lower bound>
<Side Remark line 23: upper bound>
<Side Remark line 26: find element>
<Side Remark line 30: count element>
<Side Remark line 32: erase element>

cout << "\nAfter the insert function, set1: ";
for (p = set1.begin(); p != set1.end(); p++)
    cout << "*p << ";

p = set1.lower_bound(2);
cout << "\nLower bound of 2 in set1: " << *p;
p = set1.upper_bound(2);
cout << "\nUpper bound of 2 in set1: " << *p;

p = set1.find(2);
if (p == set1.end())
    cout << "2 is not in set1" << endl;
else
    cout << "\nThe number of 2's in set1: " << set1.count(2);

set1.erase(2);
cout << "\nAfter the erase function, set1: ";
for (p = set1.begin(); p != set1.end(); p++)
    cout << "*p << ";

return 0;

<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
<end of output>

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

<Side Remark: specify an order>

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

Please note that a space after greater<int> is needed, otherwise, C++ would be confused with the >> operator.

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 24 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 all the elements with key value 2 from the set.

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

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

Trace the program with this new statement.

### 19.5.2 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 value using the key. A multimap allows duplicate keys, but a map does not. Listing 19.11 demonstrates how to use the map and multimap containers.

#### Listing 19.11 MapDemo.cpp (Using map/multimap)

```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++)
```
```cpp
    cout << p->first << " " << p->second << endl;
    cout << "Enter a key to search 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 << "After the erase function, map1:
    for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " " << p->second << endl;
    return 0; }
</output>

Initial contents in map1:
100 John Smith
101 Peter King
102 Jane Smith
103 Jeff Reed
Enter a key to search for the name: 105
Key 105 not found in map1
After the erase function, map1:
100 John Smith
101 Peter King
102 Jane Smith
<end of output>

The program creates a map map1 using its no-arg constructor (line 8), and inserts key/value pairs to map1 (lines 9-12). To insert an element to a map, you have to create a pair using the value_type(key, value) function.

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.

19.6 Container Adapters
The STL provides three container adapters: stack, queue, and priority queue. These containers 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 19.2 listed the functions common to all the containers. In addition to these common functions, each container adapter supports the push and pop functions to insert and remove an element.

19.6.1 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 19.11.

Table 19.11

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(element)</td>
<td>Inserts the element to the top of the stack.</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes an element from the top of the stack.</td>
</tr>
<tr>
<td>top()</td>
<td>Returns the top element from the stack without removing it.</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the size of the stack.</td>
</tr>
<tr>
<td>empty()</td>
<td>Returns true if the stack is empty.</td>
</tr>
</tbody>
</table>

Listing 19.12 gives an example on how to use stack.

Listing 19.12 StackDemo.cpp (Using stack)

***PD: Please add line numbers (including space lines) in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

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

#include <iostream>
#include <stack>

#include <iostream>
#include <stack>

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

<Side Remark line 2: include stack>
<Side Remark line 9: is empty?>
<Side Remark line 11: get top>
<Side Remark line 12: remove top>
<Side Remark line 18: create stack1>
<Side Remark line 19: create stack2>
<Side Remark line 23: insert to stack1>
<Side Remark line 24: insert to stack2>
<Side Remark line 28: invoke printStack>

```cpp
#include <iostream>
#include <stack>
```
This program creates a stack using the default implementation in line 18, and a stack using the vector implementation in line 19.

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.

19.6.2 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 on a queue are listed in Table 19.12.

Table 19.12
Functions in queue

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(element)</td>
<td>Inserts the element to the top of the queue.</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes an element from the top of the queue.</td>
</tr>
<tr>
<td>front()</td>
<td>Returns the front element from the queue without removing it.</td>
</tr>
<tr>
<td>back()</td>
<td>Returns the back element from the queue without removing it.</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the size of the queue.</td>
</tr>
<tr>
<td>empty()</td>
<td>Returns true if the queue is empty.</td>
</tr>
</tbody>
</table>

Listing 19.13 gives an example on how to use queue.

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

template<typename T>
void printQueue(T &queueRef)
{
    while (!queueRef.empty())
    {
        cout << queueRef.front() << " ";
        queueRef.pop();
    }
}

int main()
{
    queue<int> queue1;
    queue<int, list<int>> queue2;

    for (int i = 0; i < 8; i++)
    {
```
This program creates a queue using the default implementation in line 18, and a queue using the list implementation in line 19.

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.

19.6.2 Container Adapter: priority_queue

In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first. A priority queue has a largest-in, first-out behavior.

You can choose a vector or deque to construct a priority queue. By default, a priority queue is implemented with a vector. The priority queue class uses the same functions push, pop, top, size, and empty as in the stack class.

Listing 19.14 gives an example on how to use priority queue.

Listing 19.14 PriorityQueueDemo.cpp (Using priority_queue)

```cpp
queue1.push(i);
queue2.push(i);

cout << "Contents in queue1: ";
printQueue(queue1);

cout << "\nContents in queue2: ";
printQueue(queue2);

return 0;
```

<output>
Contents in queue1: 0 1 2 3 4 5 6 7
Contents in queue2: 0 1 2 3 4 5 6 7
<end of output>
This program creates a priority queue using the default implementation in line 18, and a priority queue using the deque implementation in line 19.

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. Since the queues are priority queues. The largest numbers are accessed and removed first.

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
• The Standard Template Library (STL) contains useful data structures. You can use them without having to reinvent the wheel.
• A container object such as a vector is used to store a collection of data, often referred to as elements.
  • 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.
  • The STL algorithms are separate from the containers. They can be applied generically on the container using iterators.
  • The sequence containers (also known as sequential containers) represent linear data structures. The three sequence containers are vector, list, and deque.
• Associative containers are non-linear 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.

• 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.

• 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.

• Iterators can be classified into five categories: input iterators, output iterators, forward iterators, bidirectional iterators, and random-access iterators.

• An input iterator is used for reading an element from a container. An output iterator is used for writing an element to a container. A forward iterator combines all the functionalities of input and output iterators to support both read and write operation. A bidirectional iterator is a forward iterator with the capability of moving backward. A random access iterator is a bidirectional iterator with the capability of accessing any element in any order.

• 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.

• 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.

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

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

Review Questions

Section 19.2 STL Basics

19.1
What are the three main components of the STL? What are the relationships among them?

19.2
What are the three types of containers? What are they used for?

19.3
Does C++ define a base class for all containers?

19.4
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 !=).

19.5
What is a first-class container?
19.6
Which containers use iterators?

19.7
Which of the following are the common features for all first-class containers?

a. Each first-class container has the \texttt{swap} function.
b. Each first-class container has the \texttt{max\_size()} function.
c. Each first-class container has the \texttt{clear()} function.
d. Each first-class container has the \texttt{erase} function.
e. Each first-class container has the \texttt{add} function.

Section 19.3 Iterators

19.8
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 next element after the last element in a container?

19.9
Show the output of the following code.

```cpp
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 << "\size " << intVector.size() << " ";
```

19.10
List the types of iterators.

Section 19.4 Sequence Containers
19.11
For what applications should you use a vector, a deque, or a list? What types of the iterators are supported in a vector, a deque, and a list?

19.12
What is wrong in the following code?

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

19.13
How do you remove elements in a vector, a deque, or a list?

19.14
Are the sort, splice, merge, and reverse functions contained in vector, deque, or list?

Section 19.5 Associative Containers

19.15
For what applications should you use a set or multiset? What are the differences between set and multiset?

19.16
Show the output of the following code.

```cpp
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 << " ";
```

19.17
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 << " ";

19.18

What is wrong in the following code?

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

19.19

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

19.20

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;

19.21

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;

19.22

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?

Section 19.6 Container Adapters

19.23

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

19.24

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

19.25

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?

Programming Exercises

19.1*
(Maximum and minimum) Implement the following functions that find the maximum and minimum elements in a first-class container.

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

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

19.2*
(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.

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

19.3*
(Occurrence of a value) Implement the following function that finds the number of occurrences of a specified value in a first-class container.

```cpp
template<typename ElementType, typename ContainerType>
int countElement(ContainerType &container, const ElementType &value)
```
(Reversing a container) Implement the following function that reverses the elements in a container.

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

19.5*

(Removing elements) Implement the following function that removes the specified values from a first-class container. The elements after the removed elements are moved to fill in for the removed elements. The total number of the elements in a container is not changed by this function.

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

19.6*

(Replacing elements) Implement the following function that replaces a given element with a new value.

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

19.7**

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

```cpp
void union(set &s1, set &s2, set &s3);
```

19.8**

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

```cpp
void difference(set &s1, set &s2, set &s3);
```

19.9**

(Displaying 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.

19.10**

(Counting 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.)

19.11**

(Counting the occurrences of numbers entered) Write a program that reads an unspecified number of integers and finds the one that has the most
occurrences. Your input ends when the input is 0. For example, if you entered 2 3 40 3 5 4 -3 3 3 2 0, the number 3 occurred most often. Please enter one number at a time. If not one but several numbers have the most occurrences, all of them should be reported. For example, since 9 and 3 appear twice in the list 9 30 3 9 3 2 4, both should be reported. (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.)

19.13**
(Counting the occurrences of words) Write a program that counts the occurrences of words in a text and displays the words and their occurrences in ascending order of word frequency. 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.