22 Standard Template Library (STL)

The shapes a bright container can contain!
—Theodore Roethke

Journey over all the universe in a map.
—Miguel de Cervantes

O! thou hast damnable iteration, and art indeed able to corrupt a saint.
—William Shakespeare

That great dust heap called "history."
—Augustine Birrell

The historian is a prophet in reverse.
—Friedrich von Schlegel

Attempt the end, and never stand to doubt; Nothing's so hard but search will find it out.
—Robert Herrick

Objectives

In this chapter you'll learn:

- To use the STL containers, container adapters and "near containers."
- To program with the dozens of STL algorithms.
- To use iterators to access the elements of STL containers.

Outline

22.1 Introduction to the Standard Template Library (STL)

We've repeatedly emphasized the importance of software reuse. Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the **Standard Template Library (STL)** to the C++ Standard Library. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. The STL offers proof of concept for generic programming with templates—introduced in Chapter 14, Templates, and used extensively in Chapter 20, Data Structures. [*Note:* In industry, the features presented in this chapter are often referred to as the Standard Template Library or STL. However, these terms are not used in the C++ standard document, because these features are simply considered to be part of the C++ Standard Library.]

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and is based on their research in the field of generic programming, with significant contributions from David Musser. As you'll see, the STL was conceived and designed for performance and flexibility.

This chapter introduces the STL and discusses its three key components—**containers** (popular templatized data structures), **iterators** and **algorithms**. The STL containers are data structures capable of storing objects of almost any data type (there are some restrictions). We'll see that there are three styles of container classes—**first-class containers**, **adapters** and **near containers**.

Performance Tip 22.1

For any particular application, several different STL containers might be appropriate. Select the most appropriate container that achieves the best performance (i.e., balance of speed and size) for that application. Efficiency was a crucial consideration in the STL's design.

Performance Tip 22.2

Standard Library capabilities are implemented to operate efficiently across many applications. For some applications with unique performance requirements, it might be necessary to write your own customized implementations.

Each STL container has associated member functions. A subset of these member functions is defined in all STL containers. We illustrate most of this common functionality in our examples of STL containers `vector` (a dynamically resizable array which we introduced in Chapter 7), **`list`** (a doubly linked list) and **`deque`** (a double-ended queue, pronounced "deck"). We introduce container-specific functionality in examples for each of the other STL containers.

STL iterators, which have properties similar to those of pointers, are used by programs to manipulate the STL-container elements. In fact, standard arrays can be manipulated by STL algorithms, using standard pointers as iterators. We'll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement. There are five categories of iterators, each of which we discuss in Section 22.1.2 and use throughout this chapter.

STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements (or entire containers). The STL provides approximately 70 algorithms. Most of them use iterators to access container elements. Each algorithm has minimum requirements for the types of iterators that can be used with it. We'll see that each first-class container supports specific iterator types, some more powerful than others. A container's supported iterator type determines whether the container can be used with a specific algorithm. Iterators encapsulate the mechanism used to access container elements. This encapsulation enables many of the STL algorithms to be applied to several containers without regard for the underlying container implementation. As long as a container's iterators support the minimum requirements of the algorithm, then the algorithm can process that container's elements. This also enables you to create new algorithms that can process the elements of multiple container types.

Software Engineering Observation 22.1

The STL approach allows general programs to be written so that the code does not depend on the underlying container. Such a programming style is called generic programming.

In Chapter 20, we studied data structures. We built linked lists, queues, stacks and trees. We carefully wove link objects together with pointers. Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints. Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work. In addition, if many programmers on a large project implement similar containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug. An advantage of the STL is that you can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations. This reuse can save substantial development time, money and effort.

Software Engineering Observation 22.2

Avoid reinventing the wheel; program with the reusable components of the C++ Standard Library. STL includes many of the most popular data structures as containers and provides various popular algorithms to process data in these containers.

Error-Prevention Tip 22.1

When programming pointer-based data structures and algorithms, we must do our own debugging and testing to be sure our data structures, classes and algorithms function properly. It's easy to make errors when manipulating pointers at this low level. Memory leaks and memory-access violations are common in such custom code. The prepackaged, templatized containers of the STL are sufficient for most programmers. Using the STL helps you reduce testing and debugging time. One caution is that, for large projects, template compile time can be significant.

This chapter introduces the STL. It's by no means complete or comprehensive. However, it's a friendly, accessible chapter that should convince you of the value of the STL in software reuse and encourage further study.

22.1.1 Introduction to Containers

The STL container types are shown in Fig. 22.1. The containers are divided into three major categories—**sequence containers**, **associative containers** and **container adapters**.

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

Fig. 22.1 | Standard Library container classes. (Part 1 of 2.)

Standard Library container class	Description
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 22.1 | Standard Library container classes. (Part 2 of 2.)

STL Containers Overview

The sequence containers represent linear data structures, such as vectors and linked lists. Associative containers are nonlinear containers that typically can locate elements stored in the containers quickly. Such containers can store sets of values or **key/value pairs**. The sequence containers and associative containers are collectively referred to as the first-class containers. As we saw in Chapter 20, stacks and queues actually are constrained versions of sequential containers. For this reason, STL implements stacks and queues as container adapters that enable a program to view a sequential container in a constrained manner. There are other container types that are considered "near containers"—C-like pointer-based arrays (discussed in Chapter 7), `bitset`s for maintaining sets of flag values and `valarray`s for performing high-speed mathematical vector operations (this last class is optimized for computation performance and is not as flexible as the first-class containers). These types are considered "near containers" because they exhibit capabilities similar to those of the first-class containers, but do not support all the first-class-container capabilities. Type `string` (discussed in Chapter 18) supports the same functionality as a sequence container, but stores only character data.

STL Container Common Functions

Most STL containers provide similar functionality. Many generic operations, such as member function `size`, apply to all containers, and other operations apply to subsets of similar containers. This encourages extensibility of the STL with new classes. Figure 22.2 describes the functions common to all Standard Library containers. [*Note:* Overloaded operators `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==` and `operator!=` are not provided for `priority_queue`s.]

Member function	Description
default constructor	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.

Fig. 22.2 | Common member functions for most STL containers. (Part 1 of 2.)

Member function	Description
`empty`	Returns `true` if there are no elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the next position after the end of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the last element of the container.
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the next position after the last element of the reversed container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 22.2 | Common member functions for most STL containers. (Part 2 of 2.)

STL Container Header Files

The header files for each of the Standard Library containers are shown in Fig. 22.3. The contents of these header files are all in `namespace std`.

First-Class Container Common `typedefs`

Figure 22.4 shows the common `typedefs` (to create synonyms or aliases for lengthy type names) found in first-class containers. These `typedefs` are used in generic declarations of variables, parameters to functions and return values from functions. For example, `value_type` in each container is always a `typedef` that represents the type of value stored in the container.

Standard Library container header files	
`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

Fig. 22.3 | Standard Library container header files.

`typedef`	Description
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference to the type of element stored in the container.
`const_reference`	A constant reference to the type of element stored in the container. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer to the type of element stored in the container.
`const_pointer`	A pointer to a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to the type of element stored in the container and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.
`const_reverse_iterator`	A constant reverse iterator that points to the type of element stored in the container and can be used only to *read* elements. This type of iterator is for iterating through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (operator `-` is not defined for iterators of `list`s and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

Fig. 22.4 | `typedef`s found in first-class containers.

Performance Tip 22.3

STL generally avoids inheritance and `virtual` functions in favor of using generic programming with templates to achieve better execution-time performance.

Portability Tip 22.1

Programming with STL will enhance the portability of your code.

When preparing to use an STL container, it's important to ensure that the type of element being stored in the container supports a minimum set of functionality. When an element is inserted into a container, a copy of that element is made. For this reason, the element type should provide its own copy constructor and assignment operator. [*Note:* This is required only if default memberwise copy and default memberwise assignment do not perform proper copy and assignment operations for the element type.] Also, the associative containers and many algorithms require elements to be compared. For this reason, the element type should provide an equality operator (==) and a less-than operator (<).

Software Engineering Observation 22.3

The STL containers do not require their elements to be comparable with the equality and less-than operators unless a program uses a container member function that must compare the container elements (e.g., the `sort` function in class `list`). Some pre-standard C++ compilers are not capable of ignoring parts of a template that are not used in a particular program. On compilers with this problem, you may not be able to use the STL containers with objects of classes that do not define overloaded less-than and equality operators.

22.1.2 Introduction to Iterators

Iterators have many features in common with pointers and are used to point to the elements of first-class containers (and for a few other purposes, as we'll see). Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, the dereferencing operator (*) dereferences an iterator so that you can use the element to which it points. The ++ operation on an iterator moves it to the next element of the container (much as incrementing a pointer into an array aims the pointer at the next element of the array).

STL first-class containers provide member functions `begin` and `end`. Function **`begin`** returns an iterator pointing to the first element of the container. Function **`end`** returns an iterator pointing to the first element past the end of the container (an element that doesn't exist). If iterator `i` points to a particular element, then `++i` points to the "next" element and `*i` refers to the element pointed to by `i`. The iterator resulting from `end` is typically used in an equality or inequality comparison to determine whether the "moving iterator" (`i` in this case) has reached the end of the container.

An object of type `iterator` refers to a container element that can be modified. An object of type `const_iterator` refers to a container element that cannot be modified.

Using `istream_iterator` for Input and Using `ostream_iterator` for Output

We use iterators with **sequences** (also called **ranges**). These sequences can be in containers, or they can be **input sequences** or **output sequences**. The program of Fig. 22.5 demonstrates input from the standard input (a sequence of data for input into a program), using an **istream_iterator**, and output to the standard output (a sequence of data for output from a program), using an **ostream_iterator**. The program inputs two integers from the user at the keyboard and displays the sum of the integers.

```
1  // Fig. 22.5: Fig22_05.cpp
2  // Demonstrating input and output with iterators.
3  #include <iostream>
4  #include <iterator> // ostream_iterator and istream_iterator
5  using namespace std;
6
7  int main()
8  {
9     cout << "Enter two integers: ";
10
11    // create istream_iterator for reading int values from cin
12    istream_iterator< int > inputInt( cin );
13
14    int number1 = *inputInt; // read int from standard input
15    ++inputInt; // move iterator to next input value
16    int number2 = *inputInt; // read int from standard input
17
18    // create ostream_iterator for writing int values to cout
19    ostream_iterator< int > outputInt( cout );
20
21    cout << "The sum is: ";
22    *outputInt = number1 + number2; // output result to cout
23    cout << endl;
24 } // end main
```

```
Enter two integers: 12 25
The sum is: 37
```

Fig. 22.5 | Input and output stream iterators.

Line 12 creates an `istream_iterator` that is capable of extracting (inputting) `int` values in a type-safe manner from the standard input object `cin`. Line 14 dereferences iterator `inputInt` to read the first integer from `cin` and assigns that integer to `number1`. The dereferencing operator `*` applied to `inputInt` gets the value from the stream associated with `inputInt`; this is similar to dereferencing a pointer. Line 15 positions iterator `inputInt` to the next value in the input stream. Line 16 inputs the next integer from `inputInt` and assigns it to `number2`.

Line 19 creates an `ostream_iterator` that is capable of inserting (outputting) `int` values in the standard output object `cout`. Line 22 outputs an integer to `cout` by assigning to `*outputInt` the sum of `number1` and `number2`. Notice the use of the dereferencing operator `*` to use `*outputInt` as an *lvalue* in the assignment statement. If you want to output another value using `outputInt`, the iterator must be incremented with `++` (both the prefix and postfix increment can be used, but the prefix form should be preferred for performance reasons).

Error-Prevention Tip 22.2

*The * (dereferencing) operator of any `const` iterator returns a `const` reference to the container element, disallowing the use of non-`const` member functions.*

Common Programming Error 22.1

Attempting to dereference an iterator positioned outside its container is a runtime logic error. In particular, the iterator returned by `end` cannot be dereferenced or incremented.

Common Programming Error 22.2

Attempting to create a non-`const` iterator for a `const` container results in a compilation error.

Iterator Categories and Iterator Category Hierarchy

Figure 22.6 shows the categories of STL iterators. Each category provides a specific set of functionality. Figure 22.7 illustrates the hierarchy of iterator categories. As you follow the hierarchy from top to bottom, each iterator category supports all the functionality of the categories above it in the figure. Thus the "weakest" iterator types are at the top and the most powerful one is at the bottom. Note that this is not an inheritance hierarchy.

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Fig. 22.6 | Iterator categories.

The iterator category that each container supports determines whether that container can be used with specific algorithms in the STL. Containers that support random-access iterators can be used with all algorithms in the STL. As we'll see, pointers into arrays can be used in place of iterators in most STL algorithms, including those that require random-access iterators. Figure 22.8 shows the iterator category of each of the STL containers. The first-class containers (`vector`s, `deque`s, `list`s, `set`s, `multiset`s, `map`s and `multimap`s), `string`s and arrays are all traversable with iterators.

Software Engineering Observation 22.4

Using the "weakest iterator" that yields acceptable performance helps produce maximally reusable components. For example, if an algorithm requires only forward iterators, it can be used with any container that supports forward iterators, bidirectional iterators or random-access iterators. However, an algorithm that requires random-access iterators can be used only with containers that have random-access iterators.

```
input       output
     \     /
     forward
        |
  bidirectional
        |
  random access
```

Fig. 22.7 | Iterator category hierarchy.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Fig. 22.8 | Iterator types supported by each container.

Predefined Iterator `typedef`*s*

Figure 22.9 shows the predefined iterator `typedef`s that are found in the class definitions of the STL containers. Not every `typedef` is defined for every container. We use `const` versions of the iterators for traversing read-only containers. We use reverse iterators to traverse containers in the reverse direction.

Predefined `typedef`s for iterator types	Direction of `++`	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

Fig. 22.9 | Iterator `typedef`s.

Error-Prevention Tip 22.3

Operations performed on a `const_iterator` *return* `const` *references to prevent modification to elements of the container being manipulated. Using* `const_iterator`*s where appropriate is another example of the principle of least privilege.*

Iterator Operations

Figure 22.10 shows some operations that can be performed on each iterator type. The operations for each iterator type include all operations preceding that type in the figure. For input iterators and output iterators, it's not possible to save the iterator then use the saved value later.

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 1 of 2.)

Iterator operation	Description
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 2 of 2.)

22.1.3 Introduction to Algorithms

STL algorithms can be used generically across a variety of containers. STL provides many algorithms you'll use frequently to manipulate containers. Inserting, deleting, searching, sorting and others are appropriate for some or all of the STL containers.

The STL includes approximately 70 standard algorithms. We show most of these and summarize the others. The algorithms operate on container elements only indirectly through iterators. Many algorithms operate on sequences of elements defined by pairs of iterators—one pointing to the first element of the sequence and one pointing to one element past the last element. Also, it's possible to create your own new algorithms that operate in a similar fashion so they can be used with the STL containers and iterators.

Algorithms often return iterators that indicate the results of the algorithms. Algorithm `find`, for example, locates an element and returns an iterator to that element. If the element is not found, `find` returns the "one past the `end`" iterator that was passed in to define the end of the range to be searched, which can be tested to determine whether an element was not found. The `find` algorithm can be used with any first-class STL container. STL algorithms create yet another opportunity for reuse—using the rich collection of popular algorithms can save you much time and effort.

If an algorithm uses less powerful iterators, the algorithm can also be used with containers that support more powerful iterators. Some algorithms demand powerful iterators; e.g., `sort` demands random-access iterators.

Software Engineering Observation 22.5

The STL is extensible. It's straightforward to add new algorithms and to do so without changes to STL containers.

Software Engineering Observation 22.6

The STL is implemented concisely. The algorithms are separated from the containers and operate on elements of the containers only indirectly through iterators. This separation makes it easier to write generic algorithms applicable to many container classes.

Software Engineering Observation 22.7

STL algorithms can operate on STL containers and on pointer-based, C-like arrays.

Portability Tip 22.2

Because STL algorithms process containers only indirectly through iterators, one algorithm can often be used with many different containers.

Figure 22.11 shows many of the **mutating-sequence algorithms**—i.e., the algorithms that result in modifications of the containers to which the algorithms are applied.

Mutating-sequence algorithms			
copy	partition	replace_copy	stable_partition
copy_backward	random_shuffle	replace_copy_if	swap
fill	remove	replace_if	swap_ranges
fill_n	remove_copy	reverse	transform
generate	remove_copy_if	reverse_copy	unique
generate_n	remove_if	rotate	unique_copy
iter_swap	replace	rotate_copy	

Fig. 22.11 | Mutating-sequence algorithms.

Figure 22.12 shows many of the nonmodifying sequence algorithms—i.e., the algorithms that do not result in modifications of the containers to which they're applied. Figure 22.13 shows the numerical algorithms of the header file **<numeric>**.

Nonmodifying sequence algorithms			
adjacent_find	equal	find_end	mismatch
count	find	find_first_of	search
count_if	find_each	find_if	search_n

Fig. 22.12 | Nonmodifying sequence algorithms.

Numerical algorithms from header file <numeric>	
accumulate	partial_sum
inner_product	adjacent_difference

Fig. 22.13 | Numerical algorithms from header file <numeric>.

22.2 Sequence Containers

The C++ Standard Template Library provides three sequence containers—`vector`, `list` and `deque`. Class template `vector` and class template `deque` both are based on arrays. Class template `list` implements a linked-list data structure similar to our `List` class presented in Chapter 20, but more robust.

One of the most popular containers in the STL is `vector`. Recall that we introduced class template `vector` in Chapter 7 as a more robust type of array. A `vector` changes size dynamically. Unlike C and C++ "raw" arrays (see Chapter 7), `vector`s can be assigned to one another. This is not possible with pointer-based, C-like arrays, because those array names are constant pointers and cannot be the targets of assignments. Just as with C arrays, `vector` subscripting does not perform automatic range checking, but class template `vector` does provide this capability via member function `at` (also discussed in Chapter 7).

Performance Tip 22.4

Insertion at the back of a `vector` is efficient. The `vector` simply grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a `vector`—the entire portion of the `vector` after the insertion (or deletion) point must be moved, because `vector` elements occupy contiguous cells in memory just as C or C++ "raw" arrays do.

Figure 22.2 presented the operations common to all the STL containers. Beyond these operations, each container typically provides a variety of other capabilities. Many of these capabilities are common to several containers, but they're not always equally efficient for each container. You must choose the container most appropriate for the application.

Performance Tip 22.5

Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` rather than a `vector`. Although we can insert and delete elements at the front and back of both a `vector` and a `deque`, class `deque` is more efficient than `vector` for doing insertions and deletions at the front.

Performance Tip 22.6

Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a `list`, due to its efficient implementation of insertion and deletion anywhere in the data structure.

In addition to the common operations described in Fig. 22.2, the sequence containers have several other common operations—**`front`** to return a reference to the first element in a non-empty container, **`back`** to return a reference to the last element in a non-empty container, `push_back` to insert a new element at the end of the container and `pop_back` to remove the last element of the container.

22.2.1 `vector` Sequence Container

Class template `vector` provides a data structure with contiguous memory locations. This enables efficient, direct access to any element of a vector via the subscript operator `[]`, exactly as with a C or C++ "raw" array. Class template `vector` is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted. When a `vector`'s memory is exhausted, the `vector` allocates a larger contiguous area of memory, copies the original elements into the new memory and deallocates the old memory.

Performance Tip 22.7

Choose the `vector` container for the best random-access performance.

Performance Tip 22.8

Objects of class template `vector` provide rapid indexed access with the overloaded subscript operator `[]` because they're stored in contiguous memory like a C or C++ raw array.

Performance Tip 22.9

It's faster to insert many elements at once than one at a time.

An important part of every container is the type of iterator it supports. This determines which algorithms can be applied to the container. A `vector` supports random-access iterators—i.e., all iterator operations shown in Fig. 22.10 can be applied to a `vector` iterator. All STL algorithms can operate on a `vector`. The iterators for a `vector` are sometimes implemented as pointers to elements of the `vector`. Each STL algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality. If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that provides forward iterators, bidirectional iterators or random-access iterators. As long as the container supports the algorithm's minimum iterator functionality, the algorithm can operate on the container.

Using Vector and Iterators

Figure 22.14 illustrates several functions of the `vector` class template. Many of these functions are available in every first-class container. You must include header file `<vector>` to use class template `vector`.

```
// Fig. 22.14: Fig22_14.cpp
// Demonstrating Standard Library vector class template.
#include <iostream>
#include <vector> // vector class-template definition
using namespace std;

// prototype for function template printVector
template < typename T > void printVector( const vector< T > &integers2 );

int main()
{
   const int SIZE = 6; // define array size
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
   vector< int > integers; // create vector of ints
```

Fig. 22.14 | Standard Library `vector` class template. (Part 1 of 2.)

```

   cout << "The initial size of integers is: " << integers.size()
      << "\nThe initial capacity of integers is: " << integers.capacity();

   // function push_back is in every sequence collection
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );

   cout << "\nThe size of integers is: " << integers.size()
      << "\nThe capacity of integers is: " << integers.capacity();
   cout << "\n\nOutput array using pointer notation: ";

   // display array using pointer notation
   for ( int *ptr = array; ptr != array + SIZE; ptr++ )
      cout << *ptr << ' ';

   cout << "\nOutput vector using iterator notation: ";
   printVector( integers );
   cout << "\nReversed contents of vector integers: ";

   // two const reverse iterators
   vector< int >::const_reverse_iterator reverseIterator;
   vector< int >::const_reverse_iterator tempIterator = integers.rend();

   // display vector in reverse order using reverse_iterator
   for ( reverseIterator = integers.rbegin();
      reverseIterator!= tempIterator; ++reverseIterator )
      cout << *reverseIterator << ' ';

   cout << endl;
} // end main

// function template for outputting vector elements
template < typename T > void printVector( const vector< T > &integers2 )
{
   typename vector< T >::const_iterator constIterator; // const_iterator

   // display vector elements using const_iterator
   for ( constIterator = integers2.begin();
      constIterator != integers2.end(); ++constIterator )
      cout << *constIterator << ' ';
} // end function printVector
```

```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Fig. 22.14 | Standard Library `vector` class template. (Part 2 of 2.)

Line 14 defines an instance called `integers` of class template `vector` that stores `int` values. When this object is instantiated, an empty `vector` is created with size 0 (i.e., the number of elements stored in the `vector`) and capacity 0 (i.e., the number of elements that can be stored without allocating more memory to the `vector`).

Lines 16 and 17 demonstrate the `size` and `capacity` functions; each initially returns 0 for `vector v` in this example. Function `size`—available in every container—returns the number of elements currently stored in the container. Function **`capacity`** returns the number of elements that can be stored in the `vector` before the `vector` needs to dynamically resize itself to accommodate more elements.

Lines 20–22 use function **`push_back`**—available in all sequence containers—to add an element to the end of the `vector`. If an element is added to a full `vector`, the `vector` increases its size—some STL implementations have the `vector` double its capacity.

Performance Tip 22.10

It can be wasteful to double a `vector`'s size when more space is needed. For example, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added. This leaves 999,999 unused elements. You can use `resize` and `reserve` to control space usage better.

Lines 24 and 25 use `size` and `capacity` to illustrate the new size and capacity of the `vector` after the three `push_back` operations. Function `size` returns 3—the number of elements added to the `vector`. Function `capacity` returns 4, indicating that we can add one more element before the `vector` needs to add more memory. When we added the first element, the `vector` allocated space for one element, and the size became 1 to indicate that the `vector` contained only one element. When we added the second element, the capacity doubled to 2 and the size became 2 as well. When we added the third element, the capacity doubled again to 4. So we can actually add another element before the `vector` needs to allocate more space. When the `vector` eventually fills its allocated capacity and the program attempts to add one more element to the `vector`, the `vector` will double its capacity to 8 elements.

The manner in which a `vector` grows to accommodate more elements—a time consuming operation—is not specified by the C++ Standard Document. C++ library implementors use various clever schemes to minimize the overhead of resizing a `vector`. Hence, the output of this program may vary, depending on the version of `vector` that comes with your compiler. Some library implementors allocate a large initial capacity. If a `vector` stores a small number of elements, such capacity may be a waste of space. However, it can greatly improve performance if a program adds many elements to a `vector` and does not have to reallocate memory to accommodate those elements. This is a classic space–time trade-off. Library implementors must balance the amount of memory used against the amount of time required to perform various `vector` operations.

Lines 29–30 demonstrate how to output the contents of an array using pointers and pointer arithmetic. Line 33 calls function `printVector` (defined in lines 49–57) to output the contents of a `vector` using iterators. Function template `printVector` receives a `const` reference to a `vector` (`integers2`) as its argument. Line 51 defines a `const_iterator` called `constIterator` that iterates through the `vector` and outputs its contents. Notice that the declaration in line 51 is prefixed with the keyword `typename`. Because `printVector` is a function template and `vector< T >` will be specialized differently for each function-template specialization, the compiler cannot tell at compile time whether or not `vector< T >::const_iterator` is a type. In a particular specialization, `const_iterator` could be a `static` variable. The compiler needs this information to compile the program correctly. Therefore, you must tell the compiler that a qualified name, when the qualifier is a dependent type, is expected to be a type in every specialization.

A `const_iterator` enables the program to read the elements of the `vector`, but does not allow the program to modify the elements. The `for` statement in lines 54–56 initializes `constIterator` using `vector` member function `begin`, which returns a `const_iterator` to the first element in the `vector`—there is another version of `begin` that returns an `iterator` that can be used for non-`const` containers. A `const_iterator` is returned because the identifier `integers2` was declared `const` in the parameter list of function `printVector`. The loop continues as long as `constIterator` has not reached the end of the `vector`. This is determined by comparing `constIterator` to the result of `integers2.end()`, which returns an iterator indicating the location past the last element of the `vector`. If `constIterator` is equal to this value, the end of the `vector` has been reached. Functions `begin` and `end` are available for all first-class containers. The body of the loop dereferences iterator `constIterator` to get the value in the current element of the `vector`. Remember that the iterator acts like a pointer to the element and that operator `*` is overloaded to return a reference to the element. The expression `++constIterator` (line 55) positions the iterator to the next element of the `vector`.

Performance Tip 22.11

Use prefix increment when applied to STL iterators because the prefix increment operator does not return a value that must be stored in a temporary object.

Error-Prevention Tip 22.4

Only random-access iterators support <. It's better to use `!=` *and* `end` *to test for the end of a container.*

Line 37 declares a `const_reverse_iterator` that can be used to iterate through a `vector` backward. Line 38 declares a `const_reverse_iterator` variable `tempIterator` and initializes it to the iterator returned by function **`rend`** (i.e., the iterator for the ending point when iterating through the container in reverse). All first-class containers support this type of iterator. Lines 41–43 use a `for` statement similar to that in function `printVector` to iterate through the `vector`. In this loop, function **`rbegin`** (i.e., the iterator for the starting point when iterating through the container in reverse) and `tempIterator` delineate the range of elements to output. As with functions `begin` and `end`, `rbegin` and `rend` can return a `const_reverse_iterator` or a `reverse_iterator`, based on whether or not the container is constant.

Performance Tip 22.12

For performance reasons, capture the loop ending value before the loop and compare against that, rather than having a (potentially expensive) function call for each iteration.

Vector Element-Manipulation Functions

Figure 22.15 illustrates functions that enable retrieval and manipulation of the elements of a `vector`. Line 15 uses an overloaded `vector` constructor that takes two iterators as arguments to initialize `integers`. Remember that pointers into an array can be used as iterators. Line 15 initializes `integers` with the contents of `array` from location `array` up to—but not including—location `array + SIZE`.

```
1  // Fig. 22.15: Fig22_15.cpp
2  // Testing Standard Library vector class template
3  // element-manipulation functions.
4  #include <iostream>
5  #include <vector> // vector class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator iterator
8  #include <stdexcept> // out_of_range exception
9  using namespace std;
10
11 int main()
12 {
13    const int SIZE = 6;
14    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15    vector< int > integers( array, array + SIZE );
16    ostream_iterator< int > output( cout, " " );
17
18    cout << "Vector integers contains: ";
19    copy( integers.begin(), integers.end(), output );
20
21    cout << "\nFirst element of integers: " << integers.front()
22       << "\nLast element of integers: " << integers.back();
23
24    integers[ 0 ] = 7; // set first element to 7
25    integers.at( 2 ) = 10; // set element at position 2 to 10
26
27    // insert 22 as 2nd element
28    integers.insert( integers.begin() + 1, 22 );
29
30    cout << "\n\nContents of vector integers after changes: ";
31    copy( integers.begin(), integers.end(), output );
32
33    // access out-of-range element
34    try
35    {
36       integers.at( 100 ) = 777;
37    } // end try
38    catch ( out_of_range &outOfRange ) // out_of_range exception
39    {
40       cout << "\n\nException: " << outOfRange.what();
41    } // end catch
42
43    // erase first element
44    integers.erase( integers.begin() );
45    cout << "\n\nVector integers after erasing first element: ";
46    copy( integers.begin(), integers.end(), output );
47
```

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 1 of 2.)

```
48      // erase remaining elements
49      integers.erase( integers.begin(), integers.end() );
50      cout << "\nAfter erasing all elements, vector integers "
51         << ( integers.empty() ? "is" : "is not" ) << " empty";
52
53      // insert elements from array
54      integers.insert( integers.begin(), array, array + SIZE );
55      cout << "\n\nContents of vector integers before clear: ";
56      copy( integers.begin(), integers.end(), output );
57
58      // empty integers; clear calls erase to empty a collection
59      integers.clear();
60      cout << "\nAfter clear, vector integers "
61         << ( integers.empty() ? "is" : "is not" ) << " empty" << endl;
62   } // end main
```

```
Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty
```

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 2 of 2.)

Line 16 defines an `ostream_iterator` called `output` that can be used to output integers separated by single spaces via `cout`. An `ostream_iterator< int >` is a type-safe output mechanism that outputs only values of type `int` or a compatible type. The first argument to the constructor specifies the output stream, and the second argument is a string specifying the separator for the values output—in this case, the string contains a space character. We use the `ostream_iterator` (defined in header `<iterator>`) to output the contents of the `vector` in this example.

Line 19 uses algorithm **`copy`** from the Standard Library to output the entire contents of `vector integers` to the standard output. Algorithm `copy` copies each element in the container starting with the location specified by the iterator in its first argument and continuing up to—but not including—the location specified by the iterator in its second argument. The first and second arguments must satisfy input iterator requirements—they must be iterators through which values can be read from a container. Also, applying `++` to the first iterator must eventually cause it to reach the second iterator argument in the container. The elements are copied to the location specified by the output iterator (i.e., an iterator through which a value can be stored or output) specified as the last argument. In this case, the output iterator is an `ostream_iterator` (`output`) that is attached to `cout`, so the elements are copied to the standard output. To use the algorithms of the Standard Library, you must include the header file **`<algorithm>`**.

Lines 21–22 use functions front and back (available for all sequence containers) to determine the vector's first and last elements, respectively. Notice the difference between functions front and begin. Function front returns a reference to the first element in the vector, while function begin returns a random access iterator pointing to the first element in the vector. Also notice the difference between functions back and end. Function back returns a reference to the last element in the vector, while function end returns a random access iterator pointing to the end of the vector (the location after the last element).

Common Programming Error 22.3

The vector must not be empty; otherwise, results of the front and back functions are undefined.

Lines 24–25 illustrate two ways to subscript through a vector (which also can be used with the deque containers). Line 26 uses the subscript operator that is overloaded to return either a reference to the value at the specified location or a constant reference to that value, depending on whether the container is constant. Function at (line 25) performs the same operation, but with bounds checking. Function at first checks the value supplied as an argument and determines whether it's in the bounds of the vector. If not, function at throws an out_of_range exception defined in header <stdexcept> (as demonstrated in lines 34–41). Figure 22.16 shows some of the STL exception types. (The Standard Library exception types are discussed in Chapter 16, Exception Handling.)

STL exception types	Description
out_of_range	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to vector member function at.
invalid_argument	Indicates an invalid argument was passed to a function.
length_error	Indicates an attempt to create too long a container, string, etc.
bad_alloc	Indicates that an attempt to allocate memory with new (or with an allocator) failed because not enough memory was available.

Fig. 22.16 | Some STL exception types.

Line 28 uses one of the three overloaded **insert** functions provided by each sequence container. Line 28 inserts the value 22 before the element at the location specified by the iterator in the first argument. In this example, the iterator is pointing to the second element of the vector, so 22 is inserted as the second element and the original second element becomes the third element of the vector. Other versions of insert allow inserting multiple copies of the same value starting at a particular position in the container, or inserting a range of values from another container (or array), starting at a particular position in the original container.

Lines 44 and 49 use the two **erase** functions that are available in all first-class containers. Line 44 indicates that the element at the location specified by the iterator argument should be removed from the container (in this example, the element at the beginning of the vector). Line 49 specifies that all elements in the range starting with the location of the first argument up to—but not including—the location of the second argument should be erased from the container. In this example, all the elements are erased from the `vector`. Line 51 uses function **`empty`** (available for all containers and adapters) to confirm that the `vector` is empty.

Common Programming Error 22.4

Erasing an element that contains a pointer to a dynamically allocated object does not `delete` *that object; this can lead to a memory leak.*

Line 54 demonstrates the version of function `insert` that uses the second and third arguments to specify the starting location and ending location in a sequence of values (possibly from another container; in this case, from array of integers `array`) that should be inserted into the `vector`. Remember that the ending location specifies the position in the sequence after the last element to be inserted; copying is performed up to—but not including—this location.

Finally, line 59 uses function **`clear`** (found in all first-class containers) to empty the `vector`. This function calls the version of `erase` used in line 51 to empty the `vector`.

[*Note:* Other functions that are common to all containers and common to all sequence containers have not yet been covered. We'll cover most of these in the next few sections. We'll also cover many functions that are specific to each container.]

22.2.2 `list` Sequence Container

The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container. If most of the insertions and deletions occur at the ends of the container, the `deque` data structure (Section 22.2.3) provides a more efficient implementation. Class template `list` is implemented as a doubly linked list—every node in the `list` contains a pointer to the previous node in the `list` and to the next node in the `list`. This enables class template `list` to support bidirectional iterators that allow the container to be traversed both forward and backward. Any algorithm that requires input, output, forward or bidirectional iterators can operate on a `list`. Many `list` member functions manipulate the elements of the container as an ordered set of elements.

In addition to the member functions of all STL containers in Fig. 22.2 and the common member functions of all sequence containers discussed in Section 22.2, class template `list` provides nine other member functions—`splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`. Several of these member functions are `list`-optimized implementations of STL algorithms presented in Section 22.5. Figure 22.17 demonstrates several features of class `list`. Remember that many of the functions presented in Figs. 22.14–22.15 can be used with class `list`. Header file **`<list>`** must be included to use class `list`.

```
// Fig. 22.17: Fig22_17.cpp
// Standard library list class template test program.
#include <iostream>
#include <list> // list class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator
using namespace std;
```

Fig. 22.17 | Standard Library `list` class template. (Part 1 of 4.)

```
// prototype for function template printList
template < typename T > void printList( const list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values; // create list of ints
   list< int > otherValues; // create list of ints

   // insert items in values
   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

   cout << "values contains: ";
   printList( values );

   values.sort(); // sort values
   cout << "\nvalues after sorting contains: ";
   printList( values );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   cout << "\nAfter insert, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert at end of values
   values.splice( values.end(), otherValues );
   cout << "\nAfter splice, values contains: ";
   printList( values );

   values.sort(); // sort values
   cout << "\nAfter sort, values contains: ";
   printList( values );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   otherValues.sort();
   cout << "\nAfter insert and sort, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert into values in sorted order
   values.merge( otherValues );
   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front(); // remove element from front
   values.pop_back(); // remove element from back
```

Fig. 22.17 | Standard Library `list` class template. (Part 2 of 4.)

```
   cout << "\nAfter pop_front and pop_back:\n   values contains: "
   printList( values );

   values.unique(); // remove duplicate elements
   cout << "\nAfter unique, values contains: ";
   printList( values );

   // swap elements of values and otherValues
   values.swap( otherValues );
   cout << "\nAfter swap:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   // replace contents of values with elements of otherValues
   values.assign( otherValues.begin(), otherValues.end() );
   cout << "\nAfter assign, values contains: ";
   printList( values );

   // remove otherValues elements and insert into values in sorted order
   values.merge( otherValues );
   cout << "\nAfter merge, values contains: ";
   printList( values );

   values.remove( 4 ); // remove all 4s
   cout << "\nAfter remove( 4 ), values contains: ";
   printList( values );
   cout << endl;
} // end main

// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template < typename T > void printList( const list< T > &listRef )
{
   if ( listRef.empty() ) // list is empty
      cout << "List is empty";
   else
   {
      ostream_iterator< T > output( cout, " " );
      copy( listRef.begin(), listRef.end(), output );
   } // end else
} // end function printList
```

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
```

Fig. 22.17 | Standard Library `list` class template. (Part 3 of 4.)

```
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ), values contains: 2 2 3 3 6 6 8 8
```

Fig. 22.17 | Standard Library `list` class template. (Part 4 of 4.)

Lines 16–17 instantiate two `list` objects capable of storing integers. Lines 20–21 use function **`push_front`** to insert integers at the beginning of `values`. Function `push_front` is specific to classes `list` and `deque` (not to `vector`). Lines 22–23 use function `push_back` to insert integers at the end of `values`. Remember that function `push_back` is common to all sequence containers.

Line 28 uses `list` member function **`sort`** to arrange the elements in the `list` in ascending order. [*Note:* This is different from the `sort` in the STL algorithms.] A second version of function `sort` allows you to supply a binary predicate function that takes two arguments (values in the list), performs a comparison and returns a `bool` value indicating the result. This function determines the order in which the elements of the `list` are sorted. This version could be particularly useful for a `list` that stores pointers rather than values. [*Note:* We demonstrate a unary predicate function in Fig. 22.28. A unary predicate function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.]

Line 38 uses `list` function **`splice`** to remove the elements in `otherValues` and insert them into `values` before the iterator position specified as the first argument. There are two other versions of this function. Function `splice` with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument. Function `splice` with four arguments uses the last two arguments to specify a range of locations that should be removed from the container in the second argument and placed at the location specified in the first argument.

After inserting more elements in `otherValues` and sorting both `values` and `otherValues`, line 53 uses `list` member function **`merge`** to remove all elements of `otherValues` and insert them in sorted order into `values`. Both `list`s must be sorted in the same order before this operation is performed. A second version of `merge` enables you to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value. The predicate function specifies the sorting order used by `merge`.

Line 59 uses `list` function **`pop_front`** to remove the first element in the `list`. Line 60 uses function **`pop_back`** (available for all sequence containers) to remove the last element in the `list`.

Line 64 uses `list` function **`unique`** to remove duplicate elements in the `list`. The `list` should be in sorted order (so that all duplicates are side by side) before this operation is performed, to guarantee that all duplicates are eliminated. A second version of `unique` enables you to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value specifying whether two elements are equal.

Line 69 uses function **`swap`** (available to all first-class containers) to exchange the contents of `values` with the contents of `otherValues`.

Line 76 uses `list` function **`assign`** (available to all sequence containers) to replace the contents of `values` with the contents of `otherValues` in the range specified by the two iterator arguments. A second version of `assign` replaces the original contents with copies of the value specified in the second argument. The first argument of the function specifies the number of copies. Line 85 uses `list` function **`remove`** to delete all copies of the value `4` from the `list`.

22.2.3 deque Sequence Container

Class `deque` provides many of the benefits of a `vector` and a `list` in one container. The term `deque` is short for "double-ended queue." Class `deque` is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a `vector`. Class `deque` is also implemented for efficient insertion and deletion operations at its front and back, much like a `list` (although a `list` is also capable of efficient insertions and deletions in the middle of the `list`). Class `deque` provides support for random-access iterators, so `deque`s can be used with all STL algorithms. One of the most common uses of a `deque` is to maintain a first-in, first-out queue of elements. In fact, a `deque` is the default underlying implementation for the `queue` adaptor (Section 22.4.2).

Additional storage for a `deque` can be allocated at either end of the `deque` in blocks of memory that are typically maintained as an array of pointers to those blocks.[1] Due to the noncontiguous memory layout of a `deque`, a `deque` iterator must be more intelligent than the pointers that are used to iterate through `vector`s or pointer-based arrays.

Performance Tip 22.13

In general, `deque` has higher overhead than `vector`.

Performance Tip 22.14

Insertions and deletions in the middle of a `deque` are optimized to minimize the number of elements copied, so it's more efficient than a `vector` but less efficient than a `list` for this kind of modification.

Class `deque` provides the same basic operations as class `vector`, but like `list` adds member functions `push_front` and `pop_front` to allow insertion and deletion at the beginning of the `deque`, respectively.

Figure 22.18 demonstrates features of class `deque`. Remember that many of the functions presented in Fig. 22.14, Fig. 22.15 and Fig. 22.17 also can be used with class `deque`. Header file **`<deque>`** must be included to use class `deque`.

Line 11 instantiates a `deque` that can store `double` values. Lines 15–17 use functions `push_front` and `push_back` to insert elements at the beginning and end of the `deque`.

1. This is an implementation-specific detail, not a requirement of the C++ standard.

Remember that `push_back` is available for all sequence containers, but `push_front` is available only for class `list` and class `deque`.

```
1 // Fig. 22.18: Fig22_18.cpp
2 // Standard Library class deque test program.
3 #include <iostream>
4 #include <deque> // deque class-template definition
5 #include <algorithm> // copy algorithm
6 #include <iterator> // ostream_iterator
7 using namespace std;
8
9 int main()
10 {
11    deque< double > values; // create deque of doubles
12    ostream_iterator< double > output( cout, " " );
13
14    // insert elements in values
15    values.push_front( 2.2 );
16    values.push_front( 3.5 );
17    values.push_back( 1.1 );
18
19    cout << "values contains: ";
20
21    // use subscript operator to obtain elements of values
22    for ( unsigned int i = 0; i < values.size(); i++ )
23       cout << values[ i ] << ' ';
24
25    values.pop_front(); // remove first element
26    cout << "\nAfter pop_front, values contains: ";
27    copy( values.begin(), values.end(), output );
28
29    // use subscript operator to modify element at location 1
30    values[ 1 ] = 5.4;
31    cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
32    copy( values.begin(), values.end(), output );
33    cout << endl;
34 } // end main
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[ 1 ] = 5.4, values contains: 2.2 5.4
```

Fig. 22.18 | Standard Library `deque` class template.

The `for` statement in lines 22–23 uses the subscript operator to retrieve the value in each element of the `deque` for output. The condition uses function `size` to ensure that we do not attempt to access an element outside the bounds of the `deque`.

Line 25 uses function `pop_front` to demonstrate removing the first element of the `deque`. Remember that `pop_front` is available only for class `list` and class `deque` (not for class `vector`).

Line 30 uses the subscript operator to create an *lvalue*. This enables values to be assigned directly to any element of the `deque`.

22.3 Associative Containers

The STL's associative containers provide direct access to store and retrieve elements via **keys** (often called **search keys**). The four associative containers are multiset, set, multimap and map. Each associative container maintains its keys in sorted order. Iterating through an associative container traverses it in the sort order for that container. Classes **multiset** and **set** provide operations for manipulating sets of values where the values are the keys—there is not a separate value associated with each key. The primary difference between a multiset and a set is that a multiset allows duplicate keys and a set does not. Classes **multimap** and **map** provide operations for manipulating values associated with keys (these values are sometimes referred to as **mapped values**). The primary difference between a multimap and a map is that a multimap allows duplicate keys with associated values to be stored and a map allows only unique keys with associated values. In addition to the common member functions of all containers presented in Fig. 22.2, all associative containers also support several other member functions, including find, lower_bound, upper_bound and count. Examples of each of the associative containers and the common associative container member functions are presented in the next several subsections.

22.3.1 multiset Associative Container

The multiset associative container provides fast storage and retrieval of keys and allows duplicate keys. The ordering of the elements is determined by a **comparator function object**. For example, in an integer multiset, elements can be sorted in ascending order by ordering the keys with **comparator function object less<int>**. We discuss function objects in detail in Section 22.7. The data type of the keys in all associative containers must support comparison properly based on the comparator function object specified—keys sorted with less< T > must support comparison with operator<. If the keys used in the associative containers are of user-defined data types, those types must supply the appropriate comparison operators. A multiset supports bidirectional iterators (but not random-access iterators).

Figure 22.19 demonstrates the multiset associative container for a multiset of integers sorted in ascending order. Header file **<set>** must be included to use class multiset. Containers multiset and set provide the same basic functionality.

```cpp
// Fig. 22.19: Fig22_19.cpp
// Testing Standard Library class multiset
#include <iostream>
#include <set> // multiset class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator
using namespace std;

// define short name for multiset type used in this program
typedef multiset< int, less< int > > Ims;

int main()
{
```

Fig. 22.19 | Standard Library multiset class template. (Part 1 of 3.)

```
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   Ims intMultiset; // Ims is typedef for "integer multiset"
   ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n";

   intMultiset.insert( 15 ); // insert 15 in intMultiset
   intMultiset.insert( 15 ); // insert 15 in intMultiset
   cout << "After inserts, there are " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n\n";

   // iterator that cannot be used to change element values
   Ims::const_iterator result;

   // find 15 in intMultiset; find returns iterator
   result = intMultiset.find( 15 );

   if ( result != intMultiset.end() ) // if iterator not at end
      cout << "Found value 15\n"; // found search value 15

   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );

   if ( result == intMultiset.end() ) // will be true hence
      cout << "Did not find value 20\n"; // did not find 20

   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );
   cout << "\nAfter insert, intMultiset contains:\n";
   copy( intMultiset.begin(), intMultiset.end(), output );

   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: "
      << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );

   // p represents pair of const_iterators
   pair< Ims::const_iterator, Ims::const_iterator > p;

   // use equal_range to determine lower and upper bound
   // of 22 in intMultiset
   p = intMultiset.equal_range( 22 );

   cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
      << *( p.first ) << "\n   Upper bound: " << *( p.second );
   cout << endl;
} // end main
```

```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset
```

Fig. 22.19 | Standard Library `multiset` class template. (Part 2 of 3.)

```
Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

Fig. 22.19 | Standard Library `multiset` class template. (Part 3 of 3.)

Line 10 uses a `typedef` to create a new type name (alias) for a `multiset` of integers ordered in ascending order, using the function object `less< int >`. Ascending order is the default for a `multiset`, so `less< int >` can be omitted in line 10. This new type (`Ims`) is then used to instantiate an integer `multiset` object, `intMultiset` (line 16).

Good Programming Practice 22.1

Use `typedef`s to make code with long type names (such as `multiset`s) easier to read.

The output statement in line 19 uses function **`count`** (available to all associative containers) to count the number of occurrences of the value `15` currently in the `multiset`.

Lines 22–23 use one of the three versions of function `insert` to add the value `15` to the `multiset` twice. A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified. A third version of `insert` takes two iterators as arguments that specify a range of values to add to the `multiset` from another container.

Line 31 uses function **`find`** (available to all associative containers) to locate the value `15` in the `multiset`. Function `find` returns an `iterator` or a `const_iterator` pointing to the earliest location at which the value is found. If the value is not found, `find` returns an `iterator` or a `const_iterator` equal to the value returned by a call to `end`. Line 40 demonstrates this case.

Line 43 uses function **`insert`** to insert the elements of array `a` into the `multiset`. In line 45, the `copy` algorithm copies the elements of the `multiset` to the standard output in ascending order.

Lines 49 and 50 use functions **`lower_bound`** and **`upper_bound`** (available in all associative containers) to locate the earliest occurrence of the value `22` in the `multiset` and the element *after* the last occurrence of the value `22` in the `multiset`. Both functions return `iterator`s or `const_iterator`s pointing to the appropriate location or the iterator returned by `end` if the value is not in the `multiset`.

Line 53 instantiates an instance of class `pair` called `p`. Objects of class `pair` are used to associate pairs of values. In this example, the contents of a `pair` are two `const_iterator`s for our integer-based `multiset`. The purpose of `p` is to store the return value of `multiset` function **`equal_range`** that returns a `pair` containing the results of both a `lower_bound` and an `upper_bound` operation. Type `pair` contains two `public` data members called **`first`** and **`second`**.

Line 57 uses function `equal_range` to determine the `lower_bound` and `upper_bound` of `22` in the `multiset`. Line 60 uses `p.first` and `p.second`, respectively, to access the `lower_bound` and `upper_bound`. We dereferenced the iterators to output the values at the locations returned from `equal_range`.

22.3.2 set Associative Container

The `set` associative container is used for fast storage and retrieval of unique keys. The implementation of a `set` is identical to that of a `multiset`, except that a `set` must have unique keys. Therefore, if an attempt is made to insert a duplicate key into a `set`, the duplicate is ignored; because this is the intended mathematical behavior of a set, we do not identify it as a common programming error. A `set` supports bidirectional iterators (but not random-access iterators). Figure 22.20 demonstrates a `set` of `double`s. Header file `<set>` must be included to use class `set`.

```
// Fig. 22.20: Fig22_20.cpp
// Standard Library class set test program.
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator> // ostream_iterator
using namespace std;

// define short name for set type used in this program
typedef set< double, less< double > > DoubleSet;

int main()
{
   const int SIZE = 5;
   double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
   DoubleSet doubleSet( a, a + SIZE );
   ostream_iterator< double > output( cout, " " );

   cout << "doubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   // p represents pair containing const_iterator and bool
   pair< DoubleSet::const_iterator, bool > p;

   // insert 13.8 in doubleSet; insert returns pair in which
   // p.first represents location of 13.8 in doubleSet and
   // p.second represents whether 13.8 was inserted
   p = doubleSet.insert( 13.8 ); // value not in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   // insert 9.5 in doubleSet
   p = doubleSet.insert( 9.5 ); // value already in set
```

Fig. 22.20 | Standard Library `set` class template. (Part 1 of 2.)

```
36      cout << "\n\n" << *( p.first )
37         << ( p.second ? " was" : " was not" ) << " inserted";
38      cout << "\ndoubleSet contains: ";
39      copy( doubleSet.begin(), doubleSet.end(), output );
40      cout << endl;
41   } // end main
```

```
doubleSet contains: 2.1 3.7 4.2 9.5

13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
```

Fig. 22.20 | Standard Library `set` class template. (Part 2 of 2.)

Line 10 uses `typedef` to create a new type name (`DoubleSet`) for a set of `double` values ordered in ascending order, using the function object `less< double >`.

Line 16 uses the new type `DoubleSet` to instantiate object `doubleSet`. The constructor call takes the elements in array `a` between `a` and `a + SIZE` (i.e., the entire array) and inserts them into the `set`. Line 20 uses algorithm `copy` to output the contents of the `set`. Notice that the value `2.1`—which appeared twice in array `a`—appears only once in `doubleSet`. This is because container `set` does not allow duplicates.

Line 23 defines a `pair` consisting of a `const_iterator` for a `DoubleSet` and a `bool` value. This object stores the result of a call to `set` function `insert`.

Line 28 uses function `insert` to place the value `13.8` in the `set`. The returned `pair`, `p`, contains an iterator `p.first` pointing to the value `13.8` in the `set` and a `bool` value that is `true` if the value was inserted and `false` if the value was not inserted (because it was already in the `set`). In this case, `13.8` was not in the set, so it was inserted. Line 35 attempts to insert `9.5`, which is already in the set. The output of lines 36–37 shows that `9.5` was not inserted.

22.3.3 `multimap` Associative Container

The `multimap` associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs). Many of the functions used with `multisets` and `sets` are also used with `multimaps` and `maps`. The elements of `multimaps` and `maps` are `pairs` of keys and values instead of individual values. When inserting into a `multimap` or `map`, a `pair` object that contains the key and the value is used. The ordering of the keys is determined by a comparator function object. For example, in a `multimap` that uses integers as the key type, keys can be sorted in ascending order by ordering them with comparator function object `less< int >`. Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is often called a one-to-many relationship. For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many students; in the military, one rank (like "private") has many people. A `multimap` supports bidirectional iterators, but not random-access iterators.

Figure 22.21 demonstrates the `multimap` associative container. Header file `<map>` must be included to use class `multimap`.

Performance Tip 22.15

A `multimap` is implemented to efficiently locate all values paired with a given key.

Line 8 uses `typedef` to define alias `Mmid` for a `multimap` type in which the key type is `int`, the type of a key's associated value is `double` and the elements are ordered in ascending order. Line 12 uses the new type to instantiate a `multimap` called `pairs`. Line 14 uses function `count` to determine the number of key/value pairs with a key of `15`.

```
1  // Fig. 22.21: Fig22_21.cpp
2  // Standard Library class multimap test program.
3  #include <iostream>
4  #include <map> // multimap class-template definition
5  using namespace std;
6
7  // define short name for multimap type used in this program
8  typedef multimap< int, double, less< int > > Mmid;
9
10 int main()
11 {
12    Mmid pairs; // declare the multimap pairs
13
14    cout << "There are currently " << pairs.count( 15 )
15       << " pairs with key 15 in the multimap\n";
16
17    // insert two value_type objects in pairs
18    pairs.insert( Mmid::value_type( 15, 2.7 ) );
19    pairs.insert( Mmid::value_type( 15, 99.3 ) );
20
21    cout << "After inserts, there are " << pairs.count( 15 )
22       << " pairs with key 15\n\n";
23
24    // insert five value_type objects in pairs
25    pairs.insert( Mmid::value_type( 30, 111.11 ) );
26    pairs.insert( Mmid::value_type( 10, 22.22 ) );
27    pairs.insert( Mmid::value_type( 25, 33.333 ) );
28    pairs.insert( Mmid::value_type( 20, 9.345 ) );
29    pairs.insert( Mmid::value_type( 5, 77.54 ) );
30
31    cout << "Multimap pairs contains:\nKey\tValue\n";
32
33    // use const_iterator to walk through elements of pairs
34    for ( Mmid::const_iterator iter = pairs.begin();
35       iter != pairs.end(); ++iter )
36       cout << iter->first << '\t' << iter->second << '\n';
37
38    cout << endl;
39 } // end main
```

Fig. 22.21 | Standard Library `multimap` class template. (Part 1 of 2.)

```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15

Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Fig. 22.21 | Standard Library `multimap` class template. (Part 2 of 2.)

Line 18 uses function `insert` to add a new key/value pair to the `multimap`. The expression `Mmid::value_type( 15, 2.7 )` creates a `pair` object in which `first` is the key (`15`) of type `int` and `second` is the value (`2.7`) of type `double`. The type `Mmid::value_type` is defined as part of the `typedef` for the `multimap`. Line 19 inserts another `pair` object with the key `15` and the value `99.3`. Then lines 21–22 output the number of pairs with key `15`.

Lines 25–29 insert five additional `pairs` into the `multimap`. The `for` statement in lines 34–36 outputs the contents of the `multimap`, including both keys and values. Line 36 uses the `const_iterator` called `iter` to access the members of the `pair` in each element of the `multimap`. Notice in the output that the keys appear in ascending order.

22.3.4 map Associative Container

The `map` associative container performs fast storage and retrieval of unique keys and associated values. Duplicate keys are not allowed—a single value can be associated with each key. This is called a **one-to-one mapping**. For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a `map` that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively. With a `map` you specify the key and get back the associated data quickly. A `map` is also known as an **associative array**. Providing the key in a `map`'s subscript operator `[]` locates the value associated with that key in the `map`. Insertions and deletions can be made anywhere in a `map`.

Figure 22.22 demonstrates a `map` and uses the same features as Fig. 22.21 to demonstrate the subscript operator. Header file `<map>` must be included to use class `map`. Lines 31–32 use the subscript operator of class `map`. When the subscript is a key that is already in the `map` (line 31), the operator returns a reference to the associated value. When the subscript is a key that is not in the `map` (line 32), the operator inserts the key in the `map` and returns a reference that can be used to associate a value with that key. Line 31 replaces the value for the key `25` (previously `33.333` as specified in line 19) with a new value, `9999.99`. Line 32 inserts a new key/value `pair` in the `map` (called **creating an association**).

```
1 // Fig. 22.22: Fig22_22.cpp
2 // Standard Library class map test program.
3 #include <iostream>
4 #include <map> // map class-template definition
```

Fig. 22.22 | Standard Library `map` class template. (Part 1 of 3.)

```cpp
using namespace std;

// define short name for map type used in this program
typedef map< int, double, less< int > > Mid;

int main()
{
   Mid pairs;

   // insert eight value_type objects in pairs
   pairs.insert( Mid::value_type( 15, 2.7 ) );
   pairs.insert( Mid::value_type( 30, 111.11 ) );
   pairs.insert( Mid::value_type( 5, 1010.1 ) );
   pairs.insert( Mid::value_type( 10, 22.22 ) );
   pairs.insert( Mid::value_type( 25, 33.333 ) );
   pairs.insert( Mid::value_type( 5, 77.54 ) ); // dup ignored
   pairs.insert( Mid::value_type( 20, 9.345 ) );
   pairs.insert( Mid::value_type( 15, 99.3 ) ); // dup ignored

   cout << "pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter = pairs.begin();
      iter != pairs.end(); ++iter )
      cout << iter->first << '\t' << iter->second << '\n';

   pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
   pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40

   cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter2 = pairs.begin();
      iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t' << iter2->second << '\n';

   cout << endl;
} // end main
```

```
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After subscript operations, pairs contains:
Key     Value
5       1010.1
10      22.22
```

Fig. 22.22 | Standard Library `map` class template. (Part 2 of 3.)

```
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

Fig. 22.22 | Standard Library `map` class template. (Part 3 of 3.)

22.4 Container Adapters

The STL provides three **container adapters**—`stack`, `queue` and `priority_queue`. Adapters are not first-class containers, because they do not provide the actual data-structure implementation in which elements can be stored and because adapters do not support iterators. The benefit of an adapter class is that you can choose an appropriate underlying data structure. All three adapter classes provide member functions **`push`** and **`pop`** that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure. The next several subsections provide examples of the adapter classes.

22.4.1 stack Adapter

Class **`stack`** enables insertions into and deletions from the underlying data structure at one end (commonly referred to as a last-in, first-out data structure). A `stack` can be implemented with any of the sequence containers: `vector`, `list` and `deque`. This example creates three integer stacks, using each of the sequence containers of the Standard Library as the underlying data structure to represent the `stack`. By default, a `stack` is implemented with a `deque`. The `stack` operations are `push` to insert an element at the top of the `stack` (implemented by calling function `push_back` of the underlying container), `pop` to remove the top element of the `stack` (implemented by calling function `pop_back` of the underlying container), **`top`** to get a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `stack` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `stack` (implemented by calling function `size` of the underlying container).

Performance Tip 22.16

Each of the common operations of a `stack` is implemented as an `inline` function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

Performance Tip 22.17

For the best performance, use class `vector` as the underlying container for a `stack`.

Figure 22.23 demonstrates the `stack` adapter class. Header file **`<stack>`** must be included to use class `stack`.

Lines 18, 21 and 24 instantiate three integer stacks. Line 18 specifies a `stack` of integers that uses the default `deque` container as its underlying data structure. Line 21 specifies

```cpp
// Fig. 22.23: Fig22_23.cpp
// Standard Library adapter stack test program.
#include <iostream>
#include <stack> // stack adapter definition
#include <vector> // vector class-template definition
#include <list> // list class-template definition
using namespace std;

// pushElements function-template prototype
template< typename T > void pushElements( T &stackRef );

// popElements function-template prototype
template< typename T > void popElements( T &stackRef );

int main()
{
   // stack with default underlying deque
   stack< int > intDequeStack;

   // stack with underlying vector
   stack< int, vector< int > > intVectorStack;

   // stack with underlying list
   stack< int, list< int > > intListStack;

   // push the values 0-9 onto each stack
   cout << "Pushing onto intDequeStack: ";
   pushElements( intDequeStack );
   cout << "\nPushing onto intVectorStack: ";
   pushElements( intVectorStack );
   cout << "\nPushing onto intListStack: ";
   pushElements( intListStack );
   cout << endl << endl;

   // display and remove elements from each stack
   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );
   cout << endl;
} // end main

// push elements onto stack object to which stackRef refers
template< typename T > void pushElements( T &stackRef )
{
   for ( int i = 0; i < 10; i++ )
   {
      stackRef.push( i ); // push element onto stack
      cout << stackRef.top() << ' '; // view (and display) top element
   } // end for
} // end function pushElements
```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 1 of 2.)

```
54
55 // pop elements from stack object to which stackRef refers
56 template< typename T > void popElements( T &stackRef )
57 {
58    while ( !stackRef.empty() )
59    {
60       cout << stackRef.top() << ' '; // view (and display) top element
61       stackRef.pop(); // remove top element
62    } // end while
63 } // end function popElements
```

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 2 of 2.)

a `stack` of integers that uses a `vector` of integers as its underlying data structure. Line 24 specifies a `stack` of integers that uses a `list` of integers as its underlying data structure.

Function `pushElements` (lines 46–53) pushes the elements onto each `stack`. Line 50 uses function `push` (available in each adapter class) to place an integer on top of the `stack`. Line 51 uses `stack` function `top` to retrieve the top element of the `stack` for output. Function `top` does not remove the top element.

Function `popElements` (lines 56–63) pops the elements off each `stack`. Line 60 uses `stack` function `top` to retrieve the top element of the `stack` for output. Line 61 uses function `pop` (available in each adapter class) to remove the top element of the `stack`. Function `pop` does not return a value.

22.4.2 queue Adapter

Class **`queue`** enables insertions at the back of the underlying data structure and deletions from the front (commonly referred to as a first-in, first-out data structure). A `queue` can be implemented with STL data structure `list` or `deque`. By default, a `queue` is implemented with a `deque`. The common `queue` operations are `push` to insert an element at the back of the `queue` (implemented by calling function `push_back` of the underlying container), `pop` to remove the element at the front of the `queue` (implemented by calling function `pop_front` of the underlying container), **`front`** to get a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container), **`back`** to get a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `queue` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `queue` (implemented by calling function `size` of the underlying container).

Performance Tip 22.18

For the best performance, use class `deque` *as the underlying container for a* `queue`*.*

Performance Tip 22.19

Each of the common operations of a `queue` is implemented as an `inline` function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

Figure 22.24 demonstrates the `queue` adapter class. Header file **`<queue>`** must be included to use a `queue`.

```
1  // Fig. 22.24: Fig22_24.cpp
2  // Standard Library adapter queue test program.
3  #include <iostream>
4  #include <queue> // queue adapter definition
5  using namespace std;
6
7  int main()
8  {
9     queue< double > values; // queue with doubles
10
11    // push elements onto queue values
12    values.push( 3.2 );
13    values.push( 9.8 );
14    values.push( 5.4 );
15
16    cout << "Popping from values: ";
17
18    // pop elements from queue
19    while ( !values.empty() )
20    {
21       cout << values.front() << ' '; // view front element
22       values.pop(); // remove element
23    } // end while
24
25    cout << endl;
26 } // end main
```

```
Popping from values: 3.2 9.8 5.4
```

Fig. 22.24 | Standard Library `queue` adapter class templates.

Line 9 instantiates a `queue` that stores `double` values. Lines 12–14 use function `push` to add elements to the `queue`. The `while` statement in lines 19–23 uses function `empty` (available in all containers) to determine whether the `queue` is empty (line 19). While there are more elements in the `queue`, line 21 uses `queue` function `front` to read (but not remove) the first element in the `queue` for output. Line 22 removes the first element in the `queue` with function `pop` (available in all adapter classes).

22.4.3 `priority_queue` Adapter

Class **`priority_queue`** provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. A `priority_queue` can be implemented with STL sequence containers `vector` or deque. By default, a priority_queue is implemented with a vector as the underlying container. When elements are added to a priority_queue, they're inserted in priority order, such that the highest-priority element (i.e., the largest value) will be the first element removed from the priority_queue. This is usually accomplished by arranging the elements in a binary tree structure called a **heap** that always maintains the largest value (i.e., highest-priority element) at the front of the data structure. We discuss the STL's heap algorithms in Section 22.5.12. The comparison of elements is performed with comparator function object less< T > by default, but you can supply a different comparator.

There are several common priority_queue operations. push inserts an element at the appropriate location based on priority order of the priority_queue (implemented by calling function push_back of the underlying container, then reordering the elements using heapsort). pop removes the highest-priority element of the priority_queue (implemented by calling function pop_back of the underlying container after removing the top element of the heap). **top** gets a reference to the top element of the priority_queue (implemented by calling function front of the underlying container). empty determines whether the priority_queue is empty (implemented by calling function empty of the underlying container). size gets the number of elements in the priority_queue (implemented by calling function size of the underlying container).

Performance Tip 22.20

Each of the common operations of a priority_queue is implemented as an inline function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

Performance Tip 22.21

For the best performance, use class vector as the underlying container for a priority_queue.

Figure 22.25 demonstrates the priority_queue adapter class. Header file <queue> must be included to use class priority_queue.

```cpp
// Fig. 22.25: Fig22_25.cpp
// Standard Library adapter priority_queue test program.
#include <iostream>
#include <queue> // priority_queue adapter definition
using namespace std;

int main()
{
   priority_queue< double > priorities; // create priority_queue

   // push elements onto priorities
   priorities.push( 3.2 );
   priorities.push( 9.8 );
   priorities.push( 5.4 );

   cout << "Popping from priorities: ";
```

Fig. 22.25 | Standard Library priority_queue adapter class. (Part 1 of 2.)

```
17
18     // pop element from priority_queue
19     while ( !priorities.empty() )
20     {
21        cout << priorities.top() << ' '; // view top element
22        priorities.pop(); // remove top element
23     } // end while
24
25     cout << endl;
26  } // end main
```

```
Popping from priorities: 9.8 5.4 3.2
```

Fig. 22.25 | Standard Library `priority_queue` adapter class. (Part 2 of 2.)

Line 9 instantiates a `priority_queue` that stores `double` values and uses a `vector` as the underlying data structure. Lines 12–14 use function `push` to add elements to the `priority_queue`. The `while` statement in lines 19–23 uses function `empty` (available in all containers) to determine whether the `priority_queue` is empty (line 19). While there are more elements, line 21 uses `priority_queue` function `top` to retrieve the highest-priority element in the `priority_queue` for output. Line 22 removes the highest-priority element in the `priority_queue` with function `pop` (available in all adapter classes).

22.5 Algorithms

Until the STL, class libraries of containers and algorithms were essentially incompatible among vendors. Early container libraries generally used inheritance and polymorphism, with the associated overhead of `virtual` function calls. Early libraries built the algorithms into the container classes as class behaviors. The STL separates the algorithms from the containers. This makes it much easier to add new algorithms. With the STL, the elements of containers are accessed through iterators. The next several subsections demonstrate many of the STL algorithms.

Performance Tip 22.22

The STL is implemented for efficiency. It avoids the overhead of `virtual` function calls.

Software Engineering Observation 22.8

STL algorithms do not depend on the implementation details of the containers on which they operate. As long as the container's (or array's) iterators satisfy the requirements of the algorithm, STL algorithms can work on C-style, pointer-based arrays, on STL containers and on user-defined data structures.

Software Engineering Observation 22.9

Algorithms can be added easily to the STL without modifying the container classes.

22.5.1 `fill`, `fill_n`, `generate` and `generate_n`

Figure 22.26 demonstrates algorithms `fill`, `fill_n`, `generate` and `generate_n`. Functions **`fill`** and **`fill_n`** set every element in a range of container elements to a specific value. Functions **`generate`** and **`generate_n`** use a **generator function** to create values for every element in a range of container elements. The generator function takes no arguments and returns a value that can be placed in an element of the container.

```cpp
// Fig. 22.26: Fig22_26.cpp
// Standard Library algorithms fill, fill_n, generate and generate_n.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

char nextLetter(); // prototype of generator function

int main()
{
   vector< char > chars( 10 );
   ostream_iterator< char > output( cout, " " );
   fill( chars.begin(), chars.end(), '5' ); // fill chars with 5s

   cout << "Vector chars after filling with 5s:\n";
   copy( chars.begin(), chars.end(), output );

   // fill first five elements of chars with As
   fill_n( chars.begin(), 5, 'A' );

   cout << "\n\nVector chars after filling five elements with As:\n";
   copy( chars.begin(), chars.end(), output );

   // generate values for all elements of chars with nextLetter
   generate( chars.begin(), chars.end(), nextLetter );

   cout << "\n\nVector chars after generating letters A-J:\n";
   copy( chars.begin(), chars.end(), output );

   // generate values for first five elements of chars with nextLetter
   generate_n( chars.begin(), 5, nextLetter );

   cout << "\n\nVector chars after generating K-O for the"
      << " first five elements:\n";
   copy( chars.begin(), chars.end(), output );
   cout << endl;
} // end main

// generator function returns next letter (starts with A)
char nextLetter()
{
   static char letter = 'A';
```

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 1 of 2.)

```
45       return letter++;
46    } // end function nextLetter
```

```
Vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:
A B C D E F G H I J

Vector chars after generating K-O for the first five elements:
K L M N O F G H I J
```

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 2 of 2.)

Line 13 defines a 10-element `vector` that stores `char` values. Line 15 uses function `fill` to place the character `'5'` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`. The iterators supplied as the first and second argument must be at least forward iterators (i.e., they can be used for both input from a container and output to a container in the forward direction).

Line 21 uses function `fill_n` to place the character `'A'` in the first five elements of `vector chars`. The iterator supplied as the first argument must be at least an output iterator (i.e., it can be used for output to a container in the forward direction). The second argument specifies the number of elements to fill. The third argument specifies the value to place in each element.

Line 27 uses function `generate` to place the result of a call to generator function `nextLetter` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`. The iterators supplied as the first and second arguments must be at least forward iterators. Function `nextLetter` (lines 42–46) begins with the character `'A'` maintained in a `static` local variable. The statement in line 45 postincrements the value of `letter` and returns the old value of `letter` each time `nextLetter` is called.

Line 33 uses function `generate_n` to place the result of a call to generator function `nextLetter` in five elements of `vector chars`, starting from `chars.begin()`. The iterator supplied as the first argument must be at least an output iterator.

22.5.2 `equal`, `mismatch` and `lexicographical_compare`

Figure 22.27 demonstrates comparing sequences of values for equality using algorithms `equal`, `mismatch` and `lexicographical_compare`.

```
// Fig. 22.27: Fig22_27.cpp
// Standard Library functions equal, mismatch and lexicographical_compare.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 1 of 3.)

```

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
   vector< int > v1( a1, a1 + SIZE ); // copy of a1
   vector< int > v2( a1, a1 + SIZE ); // copy of a1
   vector< int > v3( a2, a2 + SIZE ); // copy of a2
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );
   cout << "\nVector v3 contains: ";
   copy( v3.begin(), v3.end(), output );

   // compare vectors v1 and v2 for equality
   bool result = equal( v1.begin(), v1.end(), v2.begin() );
   cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
      << " equal to vector v2.\n";

   // compare vectors v1 and v3 for equality
   result = equal( v1.begin(), v1.end(), v3.begin() );
   cout << "Vector v1 " << ( result ? "is" : "is not" )
      << " equal to vector v3.\n";

   // location represents pair of vector iterators
   pair< vector< int >::iterator, vector< int >::iterator > location;

   // check for mismatch between v1 and v3
   location = mismatch( v1.begin(), v1.end(), v3.begin() );
   cout << "\nThere is a mismatch between v1 and v3 at location "
      << ( location.first - v1.begin() ) << "\nwhere v1 contains "
      << *location.first << " and v3 contains " << *location.second
      << "\n\n";

   char c1[ SIZE ] = "HELLO";
   char c2[ SIZE ] = "BYE BYE";

   // perform lexicographical comparison of c1 and c2
   result = lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
   cout << c1 << ( result ? " is less than " :
      " is greater than or equal to " ) << c2 << endl;
} // end main
```

```
Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 2 of 3.)

```
There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v3 contains 1000

HELLO is greater than or equal to BYE BYE
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 3 of 3.)

Line 27 uses function **`equal`** to compare two sequences of values for equality. Each sequence need not necessarily contain the same number of elements—`equal` returns `false` if the sequences are not of the same length. The `== operator` (whether built-in or overloaded) performs the comparison of the elements. In this example, the elements in `vector` `v1` from `v1.begin()` up to, but not including, `v1.end()` are compared to the elements in `vector` `v2` starting from `v2.begin()`. In this example, `v1` and `v2` are equal. The three iterator arguments must be at least input iterators (i.e., they can be used for input from a sequence in the forward direction). Line 32 uses function `equal` to compare `vectors` `v1` and `v3`, which are not equal.

There is another version of function `equal` that takes a binary predicate function as a fourth parameter. The binary predicate function receives the two elements being compared and returns a `bool` value indicating whether the elements are equal. This can be useful in sequences that store objects or pointers to values rather than actual values, because you can define one or more comparisons. For example, you can compare `Employee` objects for age, social security number, or location rather than comparing entire objects. You can compare what pointers refer to rather than comparing the pointer values (i.e., the addresses stored in the pointers).

Lines 37–40 begin by instantiating a `pair` of iterators called `location` for a `vector` of integers. This object stores the result of the call to `mismatch` (line 40). Function **`mismatch`** compares two sequences of values and returns a `pair` of iterators indicating the location in each sequence of the mismatched elements. If all the elements match, the two iterators in the `pair` are equal to the last iterator for each sequence. The three iterator arguments must be at least input iterators. Line 42 determines the actual location of the mismatch in the `vectors` with the expression `location.first - v1.begin()`. The result of this calculation is the number of elements between the iterators (this is analogous to pointer arithmetic, which we studied in Chapter 8). This corresponds to the element number in this example, because the comparison is performed from the beginning of each `vector`. As with function `equal`, there is another version of function `mismatch` that takes a binary predicate function as a fourth parameter.

Line 50 uses function **`lexicographical_compare`** to compare the contents of two character arrays. This function's four iterator arguments must be at least input iterators. As you know, pointers into arrays are random-access iterators. The first two iterator arguments specify the range of locations in the first sequence. The last two specify the range of locations in the second sequence. While iterating through the sequences, the `lexicographical_compare` checks if the element in the first sequence is less than the corresponding element in the second sequence. If so, the function returns `true`. If the element in the first sequence is greater than or equal to the element in the second sequence, the function returns `false`. This function can be used to arrange sequences lexicographically. Typically, such sequences contain strings.

22.5.3 remove, remove_if, remove_copy and remove_copy_if

Figure 22.28 demonstrates removing values from a sequence with algorithms remove, remove_if, remove_copy and remove_copy_if.

```
// Fig. 22.28: Fig22_28.cpp
// Standard Library functions remove, remove_if,
// remove_copy and remove_copy_if.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

bool greater9( int ); // prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );
   vector< int > v( a, a + SIZE ); // copy of a
   vector< int >::iterator newLastElement;

   cout << "Vector v before removing all 10s:\n   ";
   copy( v.begin(), v.end(), output );

   // remove all 10s from v
   newLastElement = remove( v.begin(), v.end(), 10 );
   cout << "\nVector v after removing all 10s:\n   ";
   copy( v.begin(), newLastElement, output );

   vector< int > v2( a, a + SIZE ); // copy of a
   vector< int > c( SIZE, 0 ); // instantiate vector c
   cout << "\n\nVector v2 before removing all 10s and copying:\n   ";
   copy( v2.begin(), v2.end(), output );

   // copy from v2 to c, removing 10s in the process
   remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
   cout << "\nVector c after removing all 10s from v2:\n   ";
   copy( c.begin(), c.end(), output );

   vector< int > v3( a, a + SIZE ); // copy of a
   cout << "\n\nVector v3 before removing all elements"
      << "\ngreater than 9:\n   ";
   copy( v3.begin(), v3.end(), output );

   // remove elements greater than 9 from v3
   newLastElement = remove_if( v3.begin(), v3.end(), greater9 );
   cout << "\nVector v3 after removing all elements"
      << "\ngreater than 9:\n   ";
   copy( v3.begin(), newLastElement, output );

```

Fig. 22.28 | Algorithms remove, remove_if, remove_copy and remove_copy_if. (Part 1 of 2.)

```
49     vector< int > v4( a, a + SIZE ); // copy of a
50     vector< int > c2( SIZE, 0 ); // instantiate vector c2
51     cout << "\n\nVector v4 before removing all elements"
52        << "\ngreater than 9 and copying:\n   ";
53     copy( v4.begin(), v4.end(), output );
54
55     // copy elements from v4 to c2, removing elements greater
56     // than 9 in the process
57     remove_copy_if( v4.begin(), v4.end(), c2.begin(), greater9 );
58     cout << "\nVector c2 after removing all elements"
59        << "\ngreater than 9 from v4:\n   ";
60     copy( c2.begin(), c2.end(), output );
61     cout << endl;
62  } // end main
63
64  // determine whether argument is greater than 9
65  bool greater9( int x )
66  {
67     return x > 9;
68  } // end function greater9
```

```
Vector v before removing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v after removing all 10s:
   2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c after removing all 10s from v2:
   2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements
greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after removing all elements
greater than 9:
   2 4 6 8

Vector v4 before removing all elements
greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after removing all elements
greater than 9 from v4:
   2 4 6 8 0 0 0 0 0 0
```

Fig. 22.28 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 2 of 2.)

Line 24 uses function **`remove`** to eliminate all elements with the value `10` in the range from `v.begin()` up to, but not including, `v.end()` from `v`. The first two iterator arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the `vector` or destroy the eliminated elements, but it does move all elements that are not eliminated toward the beginning of the `vector`. The function returns an iterator positioned after the last `vector` element that was not deleted. Elements from the iterator position to the end of the `vector` have undefined values (in this example, each “undefined” position has value 0).

Line 34 uses function **remove_copy** to copy all elements that do not have the value 10 in the range from v2.begin() up to, but not including, v2.end() from v2. The elements are placed in c, starting at position c.begin(). The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into vector c. Note, in line 29, the use of the vector constructor that receives the number of elements in the vector and the initial values of those elements.

Line 44 uses function **remove_if** to delete all those elements in the range from v3.begin() up to, but not including, v3.end() from v3 for which our user-defined unary predicate function greater9 returns true. Function greater9 (defined in lines 65–68) returns true if the value passed to it's greater than 9; otherwise, it returns false. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the vector, but it does move to the beginning of the vector all elements that are not eliminated. This function returns an iterator positioned after the last element in the vector that was not deleted. All elements from the iterator position to the end of the vector have undefined values.

Line 57 uses function **remove_copy_if** to copy all those elements in the range from v4.begin() up to, but not including, v4.end() from v4 for which the unary predicate function greater9 returns true. The elements are placed in c2, starting at position c2.begin(). The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into c2.

22.5.4 replace, replace_if, replace_copy and replace_copy_if

Figure 22.29 demonstrates replacing values from a sequence using algorithms replace, replace_if, replace_copy and replace_copy_if.

```
// Fig. 22.29: Fig22_29.cpp
// Standard Library functions replace, replace_if,
// replace_copy and replace_copy_if.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator> // ostream_iterator
using namespace std;

bool greater9( int ); // predicate function prototype

int main()
{
   const int SIZE = 10;
```

Fig. 22.29 | Algorithms replace, replace_if, replace_copy and replace_copy_if. (Part 1 of 3.)

```
    int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
    ostream_iterator< int > output( cout, " " );

    vector< int > v1( a, a + SIZE ); // copy of a
    cout << "Vector v1 before replacing all 10s:\n   ";
    copy( v1.begin(), v1.end(), output );

    // replace all 10s in v1 with 100
    replace( v1.begin(), v1.end(), 10, 100 );
    cout << "\nVector v1 after replacing 10s with 100s:\n   ";
    copy( v1.begin(), v1.end(), output );

    vector< int > v2( a, a + SIZE ); // copy of a
    vector< int > c1( SIZE ); // instantiate vector c1
    cout << "\n\nVector v2 before replacing all 10s and copying:\n   ";
    copy( v2.begin(), v2.end(), output );

    // copy from v2 to c1, replacing 10s with 100s
    replace_copy( v2.begin(), v2.end(), c1.begin(), 10, 100 );
    cout << "\nVector c1 after replacing all 10s in v2:\n   ";
    copy( c1.begin(), c1.end(), output );

    vector< int > v3( a, a + SIZE ); // copy of a
    cout << "\n\nVector v3 before replacing values greater than 9:\n   ";
    copy( v3.begin(), v3.end(), output );

    // replace values greater than 9 in v3 with 100
    replace_if( v3.begin(), v3.end(), greater9, 100 );
    cout << "\nVector v3 after replacing all values greater"
       << "\nthan 9 with 100s:\n   ";
    copy( v3.begin(), v3.end(), output );

    vector< int > v4( a, a + SIZE ); // copy of a
    vector< int > c2( SIZE ); // instantiate vector c2‘
    cout << "\n\nVector v4 before replacing all values greater "
       << "than 9 and copying:\n   ";
    copy( v4.begin(), v4.end(), output );

    // copy v4 to c2, replacing elements greater than 9 with 100
    replace_copy_if( v4.begin(), v4.end(), c2.begin(), greater9, 100 );
    cout << "\nVector c2 after replacing all values greater "
       << "than 9 in v4:\n   ";
    copy( c2.begin(), c2.end(), output );
    cout << endl;
 } // end main

 // determine whether argument is greater than 9
 bool greater9( int x )
 {
    return x > 9;
 } // end function greater9
```

Fig. 22.29 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`. (Part 2 of 3.)

```
Vector v1 before replacing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing 10s with 100s:
   100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
   100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
   100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater than 9 in v4:
   100 2 100 4 100 6 100 8 100 100
```

Fig. 22.29 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`. (Part 3 of 3.)

Line 23 uses function **`replace`** to replace all elements with the value `10` in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1` with the new value `100`. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

Line 33 uses function **`replace_copy`** to copy all elements in the range from `v2.begin()` up to, but not including, `v2.end()` from `v2`, replacing all elements with the value `10` with the new value `100`. The elements are copied into `c1`, starting at position `c1.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c1`.

Line 42 uses function **`replace_if`** to replace all those elements in the range from `v3.begin()` up to, but not including, `v3.end()` in `v3` for which the unary predicate function `greater9` returns `true`. Function `greater9` (defined in lines 62–65) returns `true` if the value passed to it's greater than 9; otherwise, it returns `false`. The value `100` replaces each value greater than 9. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

Line 54 uses function **`replace_copy_if`** to copy all elements in the range from `v4.begin()` up to, but not including, `v4.end()` from `v4`. Elements for which the unary predicate function `greater9` returns `true` are replaced with the value `100`. The elements are placed in `c2`, starting at position `c2.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c2`.

22.5.5 Mathematical Algorithms

Figure 22.30 demonstrates several common mathematical algorithms from the STL, including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` and `transform`.

```
// Fig. 22.30: Fig22_30.cpp
// Mathematical algorithms of the Standard Library.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <numeric> // accumulate is defined here
#include <vector>
#include <iterator>
using namespace std;

bool greater9( int ); // predicate function prototype
void outputSquare( int ); // output square of a value
int calculateCube( int ); // calculate cube of a value

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE ); // copy of a1
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before random_shuffle: ";
   copy( v.begin(), v.end(), output );

   random_shuffle( v.begin(), v.end() ); // shuffle elements of v
   cout << "\nVector v after random_shuffle: ";
   copy( v.begin(), v.end(), output );

   int a2[ SIZE ] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   vector< int > v2( a2, a2 + SIZE ); // copy of a2
   cout << "\n\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );

   // count number of elements in v2 with value 8
   int result = count( v2.begin(), v2.end(), 8 );
   cout << "\nNumber of elements matching 8: " << result;

   // count number of elements in v2 that are greater than 9
   result = count_if( v2.begin(), v2.end(), greater9 );
   cout << "\nNumber of elements greater than 9: " << result;

   // locate minimum element in v2
   cout << "\n\nMinimum element in Vector v2 is: "
      << *( min_element( v2.begin(), v2.end() ) );

   // locate maximum element in v2
   cout << "\nMaximum element in Vector v2 is: "
      << *( max_element( v2.begin(), v2.end() ) );
```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 1 of 2.)

```cpp

   // calculate sum of elements in v
   cout << "\n\nThe total of the elements in Vector v is: "
      << accumulate( v.begin(), v.end(), 0 );

   // output square of every element in v
   cout << "\n\nThe square of every integer in Vector v is:\n";
   for_each( v.begin(), v.end(), outputSquare );

   vector< int > cubes( SIZE ); // instantiate vector cubes

   // calculate cube of each element in v; place results in cubes
   transform( v.begin(), v.end(), cubes.begin(), calculateCube );
   cout << "\n\nThe cube of every integer in Vector v is:\n";
   copy( cubes.begin(), cubes.end(), output );
   cout << endl;
} // end main

// determine whether argument is greater than 9
bool greater9( int value )
{
   return value > 9;
} // end function greater9

// output square of argument
void outputSquare( int value )
{
   cout << value * value << ' ';
} // end function outputSquare

// return cube of argument
int calculateCube( int value )
{
   return value * value * value;
} // end function calculateCube
```

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8
```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 2 of 2.)

Line 24 uses function **random_shuffle** to reorder randomly the elements in the range from v.begin() up to, but not including, v.end() in v. This function takes two random-access iterator arguments.

Line 34 uses function **count** to count the elements with the value 8 in the range from v2.begin() up to, but not including, v2.end() in v2. This function requires its two iterator arguments to be at least input iterators.

Line 38 uses function **count_if** to count elements in the range from v2.begin() up to, but not including, v2.end() in v2 for which the predicate function greater9 returns true. Function count_if requires its two iterator arguments to be at least input iterators.

Line 43 uses function **min_element** to locate the smallest element in the range from v2.begin() up to, but not including, v2.end(). The function returns a forward iterator located at the smallest element, or v2.end() if the range is empty. The function's two iterator arguments must be at least input iterators. A second version of this function takes as its third argument a binary function that compares two elements in the sequence. This function returns the bool value true if the first argument is less than the second.

Good Programming Practice 22.2

It's a good practice to check that the range specified in a call to min_element *is not empty and that the return value is not the "past the end" iterator.*

Line 47 uses function **max_element** to locate the largest element in the range from v2.begin() up to, but not including, v2.end() in v2. The function returns an input iterator located at the largest element. The function's two iterator arguments must be at least input iterators. A second version of this function takes as its third argument a binary predicate function that compares the elements in the sequence. The binary function takes two arguments and returns the bool value true if the first argument is less than the second.

Line 51 uses function **accumulate** (the template of which is in header file <numeric>) to sum the values in the range from v.begin() up to, but not including, v.end() in v. The function's two iterator arguments must be at least input iterators and its third argument represents the initial value of the total. A second version of this function takes as its fourth argument a general function that determines how elements are accumulated. The general function must take two arguments and return a result. The first argument to this function is the current value of the accumulation. The second argument is the value of the current element in the sequence being accumulated.

Line 55 uses function **for_each** to apply a general function to every element in the range from v.begin() up to, but not including, v.end(). The general function takes the current element as an argument and may modify that element (if it's received by reference). Function for_each requires its two iterator arguments to be at least input iterators.

Line 60 uses function **transform** to apply a general function to every element in the range from v.begin() up to, but not including, v.end() in v. The general function (the fourth argument) should take the current element as an argument, should not modify the element and should return the transformed value. Function transform requires its first two iterator arguments to be at least input iterators and its third argument to be at least an output iterator. The third argument specifies where the transformed values should be placed. Note that the third argument can equal the first. Another version of transform accepts five arguments—the first two arguments are input iterators that specify a range of elements from one source container, the third argument is an input iterator that specifies the first element in another source container, the fourth argument is an output iterator that specifies where the transformed values should be placed and the last argument is a general function that takes two arguments. This version of `transform` takes one element from each of the two input sources and applies the general function to that pair of elements, then places the transformed value at the location specified by the fourth argument.

22.5.6 Basic Searching and Sorting Algorithms

Figure 22.31 demonstrates some basic searching and sorting capabilities of the Standard Library, including `find`, `find_if`, `sort` and `binary_search`.

```cpp
// Fig. 22.31: Fig22_31.cpp
// Standard Library search and sort algorithms.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator>
using namespace std;

bool greater10( int value ); // predicate function prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
   vector< int > v( a, a + SIZE ); // copy of a
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output ); // display output vector

   // locate first occurrence of 16 in v
   vector< int >::iterator location;
   location = find( v.begin(), v.end(), 16 );

   if ( location != v.end() ) // found 16
      cout << "\n\nFound 16 at location " << ( location - v.begin() );
   else // 16 not found
      cout << "\n\n16 not found";

   // locate first occurrence of 100 in v
   location = find( v.begin(), v.end(), 100 );

   if ( location != v.end() ) // found 100
      cout << "\nFound 100 at location " << ( location - v.begin() );
   else // 100 not found
      cout << "\n100 not found";

   // locate first occurrence of value greater than 10 in v
   location = find_if( v.begin(), v.end(), greater10 );

```

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 1 of 2.)

```
41      if ( location != v.end() ) // found value greater than 10
42         cout << "\n\nThe first value greater than 10 is " << *location
43            << "\nfound at location " << ( location - v.begin() );
44      else // value greater than 10 not found
45         cout << "\n\nNo values greater than 10 were found";
46
47      // sort elements of v
48      sort( v.begin(), v.end() );
49      cout << "\n\nVector v after sort: ";
50      copy( v.begin(), v.end(), output );
51
52      // use binary_search to locate 13 in v
53      if ( binary_search( v.begin(), v.end(), 13 ) )
54         cout << "\n\n13 was found in v";
55      else
56         cout << "\n\n13 was not found in v";
57
58      // use binary_search to locate 100 in v
59      if ( binary_search( v.begin(), v.end(), 100 ) )
60         cout << "\n100 was found in v";
61      else
62         cout << "\n100 was not found in v";
63
64      cout << endl;
65   } // end main
66
67   // determine whether argument is greater than 10
68   bool greater10( int value )
69   {
70      return value > 10;
71   } // end function greater10
```

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 2 of 2.)

Line 23 uses function **`find`** to locate the value 16 in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The function requires its two iterator arguments to be at least input iterators and returns an input iterator that either is positioned at the first element containing the value or indicates the end of the sequence (as is the case in line 31).

Line 39 uses function **`find_if`** to locate the first value in the range from `v.begin()` up to, but not including, `v.end()` in `v` for which the unary predicate function `greater10` returns `true`. Function `greater10` (defined in lines 71–74) takes an integer and returns a bool value indicating whether the integer argument is greater than 10. Function find_if requires its two iterator arguments to be at least input iterators. The function returns an input iterator that either is positioned at the first element containing a value for which the predicate function returns true or indicates the end of the sequence.

Line 48 uses function **sort** to arrange the elements in the range from v.begin() up to, but not including, v.end() in v in ascending order. The function requires its two iterator arguments to be random-access iterators. A second version of this function takes a third argument that is a binary predicate function taking two arguments that are values in the sequence and returning a bool indicating the sorting order—if the return value is true, the two elements being compared are in sorted order.

Common Programming Error 22.5

Attempting to sort a container by using an iterator other than a random-access iterator is a compilation error. Function sort requires a random-access iterator.

Line 53 uses function **binary_search** to determine whether the value 13 is in the range from v.begin() up to, but not including, v.end() in v. The sequence of values must be sorted in ascending order first. Function binary_search requires its two iterator arguments to be at least forward iterators. The function returns a bool indicating whether the value was found in the sequence. Line 59 demonstrates a call to function binary_search in which the value is not found. A second version of this function takes a fourth argument that is a binary predicate function taking two arguments that are values in the sequence and returning a bool. The predicate function returns true if the two elements being compared are in sorted order. To obtain the location of the search key in the container, use the lower_bound or find algorithms.

22.5.7 swap, iter_swap and swap_ranges

Figure 22.32 demonstrates algorithms swap, iter_swap and swap_ranges for swapping elements. Line 18 uses function **swap** to exchange two values. In this example, the first and second elements of array a are exchanged. The function takes as arguments references to the two values being exchanged.

```
1  // Fig. 22.32: Fig22_32.cpp
2  // Standard Library algorithms iter_swap, swap and swap_ranges.
3  #include <iostream>
4  #include <algorithm> // algorithm definitions
5  #include <iterator>
6  using namespace std;
7
8  int main()
9  {
10    const int SIZE = 10;
11    int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
12    ostream_iterator< int > output( cout, " " );
13
14    cout << "Array a contains:\n   ";
15    copy( a, a + SIZE, output ); // display array a
```

Fig. 22.32 | Demonstrating swap, iter_swap and swap_ranges. (Part 1 of 2.)

```
16
17     // swap elements at locations 0 and 1 of array a
18     swap( a[ 0 ], a[ 1 ] );
19
20     cout << "\nArray a after swapping a[0] and a[1] using swap:\n   ";
21     copy( a, a + SIZE, output ); // display array a
22
23     // use iterators to swap elements at locations 0 and 1 of array a
24     iter_swap( &a[ 0 ], &a[ 1 ] ); // swap with iterators
25     cout << "\nArray a after swapping a[0] and a[1] using iter_swap:\n   ";
26     copy( a, a + SIZE, output );
27
28     // swap elements in first five elements of array a with
29     // elements in last five elements of array a
30     swap_ranges( a, a + 5, a + 5 );
31
32     cout << "\nArray a after swapping the first five elements\n"
33        << "with the last five elements:\n   ";
34     copy( a, a + SIZE, output );
35     cout << endl;
36  } // end main
```

```
Array a contains:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
   2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
   6 7 8 9 10 1 2 3 4 5
```

Fig. 22.32 | Demonstrating `swap`, `iter_swap` and `swap_ranges`. (Part 2 of 2.)

Line 24 uses function **`iter_swap`** to exchange the two elements. The function takes two forward iterator arguments (in this case, pointers to elements of an array) and exchanges the values in the elements to which the iterators refer.

Line 30 uses function **`swap_ranges`** to exchange the elements from `a` up to, but not including, `a + 5` with the elements beginning at position `a + 5`. The function requires three forward iterator arguments. The first two arguments specify the range of elements in the first sequence that will be exchanged with the elements in the second sequence starting from the iterator in the third argument. In this example, the two sequences of values are in the same array, but the sequences can be from different arrays or containers.

22.5.8 `copy_backward`, `merge`, `unique` and `reverse`

Figure 22.33 demonstrates STL algorithms `copy_backward`, `merge`, `unique` and `reverse`. Line 26 uses function **`copy_backward`** to copy elements in the range from `v1.begin()` up to, but not including, `v1.end()`, placing the elements in `results` by starting from the element before `results.end()` and working toward the beginning of the `vector`. The function returns an iterator positioned at the last element copied into the `results` (i.e., the beginning of `results`, because of the backward copy). The elements are placed in `results` in the same order as `v1`. This function requires three bidirectional iterator arguments (iterators that can be incremented and decremented to iterate forward and backward through a sequence, respectively). One difference between `copy_backward` and `copy` is that the iterator returned from `copy` is positioned *after* the last element copied and the one returned from `copy_backward` is positioned *at* the last element copied (i.e., the first element in the sequence). Also, `copy_backward` can manipulate overlapping ranges of elements in a container as long as the first element to copy is not in the destination range of elements.

```cpp
// Fig. 22.33: Fig22_33.cpp
// Standard Library functions copy_backward, merge, unique and reverse.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE = 5;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
   int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE ); // copy of a1
   vector< int > v2( a2, a2 + SIZE ); // copy of a2
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output ); // display vector output
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output ); // display vector output

   vector< int > results( v1.size() );

   // place elements of v1 into results in reverse order
   copy_backward( v1.begin(), v1.end(), results.end() );
   cout << "\n\nAfter copy_backward, results contains: ";
   copy( results.begin(), results.end(), output );

   vector< int > results2( v1.size() + v2.size() );

   // merge elements of v1 and v2 into results2 in sorted order
   merge( v1.begin(), v1.end(), v2.begin(), v2.end(), results2.begin() );

   cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
   copy( results2.begin(), results2.end(), output );

   // eliminate duplicate values from results2
   vector< int >::iterator endLocation;
   endLocation = unique( results2.begin(), results2.end() );

   cout << "\n\nAfter unique results2 contains:\n";
   copy( results2.begin(), endLocation, output );
```

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 1 of 2.)

```
44
45        cout << "\n\nVector v1 after reverse: ";
46        reverse( v1.begin(), v1.end() ); // reverse elements of v1
47        copy( v1.begin(), v1.end(), output );
48        cout << endl;
49     } // end main
```

```
Vector v1 contains: 1 3 5 7 9
Vector v2 contains: 2 4 5 7 9

After copy_backward, results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1
```

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 2 of 2.)

Line 33 uses function **`merge`** to combine two sorted ascending sequences of values into a third sorted ascending sequence. The function requires five iterator arguments. The first four must be at least input iterators and the last must be at least an output iterator. The first two arguments specify the range of elements in the first sorted sequence (`v1`), the second two arguments specify the range of elements in the second sorted sequence (`v2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged. A second version of this function takes as its sixth argument a binary predicate function that specifies the sorting order.

Line 30 creates vector `results2` with the number of elements `v1.size() + v2.size()`. Using the `merge` function as shown here requires that the sequence where the results are stored be at least the size of the two sequences being merged. If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation, you can use the following statements:

```
vector< int > results2;
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
   back_inserter( results2 ) );
```

The argument `back_inserter(results2)` uses function template **`back_inserter`** (header file `<iterator>`) for the container `results2`. A `back_inserter` calls the container's default `push_back` function to insert an element at the end of the container. If an element is inserted into a container that has no more space available, *the container grows in size*. Thus, the number of elements in the container does not have to be known in advance. There are two other inserters—**`front_inserter`** (to insert an element at the beginning of a container specified as its argument) and **`inserter`** (to insert an element before the iterator supplied as its second argument in the container supplied as its first argument).

Line 40 uses function **`unique`** on the sorted sequence of elements in the range from `results2.begin()` up to, but not including, `results2.end()` in `results2`. After this function is applied to a sorted sequence with duplicate values, only a single copy of each value remains in the sequence. The function takes two arguments that must be at least forward iterators. The function returns an iterator positioned after the last element in the sequence of unique values. The values of all elements in the container after the last unique value are undefined. A second version of this function takes as a third argument a binary predicate function specifying how to compare two elements for equality.

Line 46 uses function **reverse** to reverse all the elements in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1`. The function takes two arguments that must be at least bidirectional iterators.

22.5.9 `inplace_merge`, `unique_copy` and `reverse_copy`

Figure 22.34 demonstrates algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. Line 22 uses function **inplace_merge** to merge two sorted sequences of elements in the same container. In this example, the elements from `v1.begin()` up to, but not including, `v1.begin() + 5` are merged with the elements from `v1.begin() + 5` up to, but not including, `v1.end()`. This function requires its three iterator arguments to be at least bidirectional iterators. A second version of this function takes as a fourth argument a binary predicate function for comparing elements in the two sequences.

```
1  // Fig. 22.34: Fig22_34.cpp
2  // Standard Library algorithms inplace_merge,
3  // reverse_copy and unique_copy.
4  #include <iostream>
5  #include <algorithm> // algorithm definitions
6  #include <vector> // vector class-template definition
7  #include <iterator> // back_inserter definition
8  using namespace std;
9
10 int main()
11 {
12    const int SIZE = 10;
13    int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
14    vector< int > v1( a1, a1 + SIZE ); // copy of a
15    ostream_iterator< int > output( cout, " " );
16
17    cout << "Vector v1 contains: ";
18    copy( v1.begin(), v1.end(), output );
19
20    // merge first half of v1 with second half of v1 such that
21    // v1 contains sorted set of elements after merge
22    inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
23
24    cout << "\nAfter inplace_merge, v1 contains: ";
25    copy( v1.begin(), v1.end(), output );
26
27    vector< int > results1;
28
29    // copy only unique elements of v1 into results1
30    unique_copy( v1.begin(), v1.end(), back_inserter( results1 ) );
31    cout << "\nAfter unique_copy results1 contains: ";
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 1 of 2.)

```
32      copy( results1.begin(), results1.end(), output );
33
34      vector< int > results2;
35
36      // copy elements of v1 into results2 in reverse order
37      reverse_copy( v1.begin(), v1.end(), back_inserter( results2 ) );
38      cout << "\nAfter reverse_copy, results2 contains: ";
39      copy( results2.begin(), results2.end(), output );
40      cout << endl;
41   } // end main
```

```
Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 2 of 2.)

Line 30 uses function **`unique_copy`** to make a copy of all the unique elements in the sorted sequence of values from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are placed into `vector` `results1`. The first two arguments must be at least input iterators and the last must be at least an output iterator. In this example, we did not preallocate enough elements in `results1` to store all the elements copied from `v1`. Instead, we use function `back_inserter` (defined in header file `<iterator>`) to add elements to the end of `v1`. The `back_inserter` uses class `vector`'s capability to insert elements at the end of the `vector`. Because the `back_inserter` inserts an element rather than replacing an existing element's value, the `vector` is able to grow to accommodate additional elements. A second version of the `unique_copy` function takes as a fourth argument a binary predicate function for comparing elements for equality.

Line 37 uses function **`reverse_copy`** to make a reversed copy of the elements in the range from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are inserted into `results2` using a `back_inserter` object to ensure that the `vector` can grow to accommodate the appropriate number of elements copied. Function `reverse_copy` requires its first two iterator arguments to be at least bidirectional iterators and its third to be at least an output iterator.

22.5.10 Set Operations

Figure 22.35 demonstrates functions `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union` for manipulating sets of sorted values. To demonstrate that STL functions can be applied to arrays and containers, this example uses only arrays (remember, a pointer into an array is a random-access iterator).

Lines 25 and 31 call function **`includes`**. Function `includes` compares two sets of sorted values to determine whether every element of the second set is in the first set. If so, `includes` returns `true`; otherwise, it returns `false`. The first two iterator arguments must be at least input iterators and must describe the first set of values. In line 25, the first set consists of the elements from `a1` up to, but not including, `a1 + SIZE1`. The last two iterator arguments must be at least input iterators and must describe the second set of values. In this example, the second set consists of the elements from `a2` up to, but not including, `a2`

+ `SIZE2`. A second version of function `includes` takes a fifth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

```
// Fig. 22.35: Fig22_35.cpp
// Standard Library algorithms includes, set_difference,
// set_intersection, set_symmetric_difference and set_union.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
   int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
   int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
   ostream_iterator< int > output( cout, " " );

   cout << "a1 contains: ";
   copy( a1, a1 + SIZE1, output ); // display array a1
   cout << "\na2 contains: ";
   copy( a2, a2 + SIZE2, output ); // display array a2
   cout << "\na3 contains: ";
   copy( a3, a3 + SIZE2, output ); // display array a3

   // determine whether set a2 is completely contained in a1
   if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
      cout << "\n\na1 includes a2";
   else
      cout << "\n\na1 does not include a2";

   // determine whether set a3 is completely contained in a1
   if ( includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
      cout << "\na1 includes a3";
   else
      cout << "\na1 does not include a3";

   int difference[ SIZE1 ];

   // determine elements of a1 not in a2
   int *ptr = set_difference( a1, a1 + SIZE1,
      a2, a2 + SIZE2, difference );
   cout << "\n\nset_difference of a1 and a2 is: ";
   copy( difference, ptr, output );

   int intersection[ SIZE1 ];

   // determine elements in both a1 and a2
   ptr = set_intersection( a1, a1 + SIZE1,
      a2, a2 + SIZE2, intersection );
```

Fig. 22.35 | `set` operations of the Standard Library. (Part 1 of 2.)

```
49      cout << "\n\nset_intersection of a1 and a2 is: ";
50      copy( intersection, ptr, output );
51
52      int symmetric_difference[ SIZE1 + SIZE2 ];
53
54      // determine elements of a1 that are not in a2 and
55      // elements of a2 that are not in a1
56      ptr = set_symmetric_difference( a1, a1 + SIZE1,
57         a3, a3 + SIZE2, symmetric_difference );
58      cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
59      copy( symmetric_difference, ptr, output );
60
61      int unionSet[ SIZE3 ];
62
63      // determine elements that are in either or both sets
64      ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
65      cout << "\n\nset_union of a1 and a3 is: ";
66      copy( unionSet, ptr, output );
67      cout << endl;
68   } // end main
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15

a1 includes a2
a1 does not include a3

set_difference of a1 and a2 is: 1 2 3 9 10

set_intersection of a1 and a2 is: 4 5 6 7 8

set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15

set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

Fig. 22.35 | `set` operations of the Standard Library. (Part 2 of 2.)

Lines 39–40 use function **`set_difference`** to find the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in ascending order). The elements that are different are copied into the fifth argument (in this case, the array `difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 47–48 use function **`set_intersection`** to determine the elements from the first set of sorted values that are in the second set of sorted values (both sets of values must be in ascending order). The elements common to both sets are copied into the fifth argument (in this case, array `intersection`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are the same. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_intersection` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 56–57 use function **`set_symmetric_difference`** to determine the elements in the first set that are not in the second set and the elements in the second set that are not in the first set (both sets must be in ascending order). The elements that are different are copied from both sets into the fifth argument (the array `symmetric_difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_symmetric_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Line 64 uses function **`set_union`** to create a set of all the elements that are in either or both of the two sorted sets (both sets of values must be in ascending order). The elements are copied from both sets into the fifth argument (in this case the array `unionSet`). Elements that appear in both sets are only copied from the first set. The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store the copied elements. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of `set_union` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

22.5.11 `lower_bound`, `upper_bound` and `equal_range`

Figure 22.36 demonstrates functions `lower_bound`, `upper_bound` and `equal_range`. Line 22 uses function **`lower_bound`** to find the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the lower bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of function `lower_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

Line 28 uses function **`upper_bound`** to find the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the upper bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of `upper_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

```
// Fig. 22.36: Fig22_36.cpp
// Standard Library functions lower_bound, upper_bound and
// equal_range for a sorted sequence of values.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
   vector< int > v( a1, a1 + SIZE ); // copy of a1
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains:\n";
   copy( v.begin(), v.end(), output );

   // determine lower-bound insertion point for 6 in v
   vector< int >::iterator lower;
   lower = lower_bound( v.begin(), v.end(), 6 );
   cout << "\n\nLower bound of 6 is element "
      << ( lower - v.begin() ) << " of vector v";

   // determine upper-bound insertion point for 6 in v
   vector< int >::iterator upper;
   upper = upper_bound( v.begin(), v.end(), 6 );
   cout << "\nUpper bound of 6 is element "
      << ( upper - v.begin() ) << " of vector v";

   // use equal_range to determine both the lower- and
   // upper-bound insertion points for 6
   pair< vector< int >::iterator, vector< int >::iterator > eq;
   eq = equal_range( v.begin(), v.end(), 6 );
   cout << "\nUsing equal_range:\n   Lower bound of 6 is element "
        << ( eq.first - v.begin() ) << " of vector v";
   cout << "\n   Upper bound of 6 is element "
      << ( eq.second - v.begin() ) << " of vector v";
   cout << "\n\nUse lower_bound to locate the first point\n"
      << "at which 5 can be inserted in order";

   // determine lower-bound insertion point for 5 in v
   lower = lower_bound( v.begin(), v.end(), 5 );
   cout << "\n   Lower bound of 5 is element "
      << ( lower - v.begin() ) << " of vector v";
   cout << "\n\nUse upper_bound to locate the last point\n"
      << "at which 7 can be inserted in order";

```

Fig. 22.36 | Algorithms `lower_bound`, `upper_bound` and `equal_range`. (Part 1 of 2.)

```
50      // determine upper-bound insertion point for 7 in v
51      upper = upper_bound( v.begin(), v.end(), 7 );
52      cout << "\n   Upper bound of 7 is element "
53         << ( upper - v.begin() ) << " of vector v";
54      cout << "\n\nUse equal_range to locate the first and\n"
55         << "last point at which 5 can be inserted in order";
56
57      // use equal_range to determine both the lower- and
58      // upper-bound insertion points for 5
59      eq = equal_range( v.begin(), v.end(), 5 );
60      cout << "\n   Lower bound of 5 is element "
61         << ( eq.first - v.begin() ) << " of vector v";
62      cout << "\n   Upper bound of 5 is element "
63         << ( eq.second - v.begin() ) << " of vector v" << endl;
64   } // end main
```

```
Vector v contains:
2 2 4 4 4 6 6 6 6 8

Lower bound of 6 is element 5 of vector v
Upper bound of 6 is element 9 of vector v
Using equal_range:
   Lower bound of 6 is element 5 of vector v
   Upper bound of 6 is element 9 of vector v

Use lower_bound to locate the first point
at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v

Use upper_bound to locate the last point
at which 7 can be inserted in order
   Upper bound of 7 is element 9 of vector v

Use equal_range to locate the first and
last point at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v
   Upper bound of 5 is element 5 of vector v
```

Fig. 22.36 | Algorithms `lower_bound`, `upper_bound` and `equal_range`. (Part 2 of 2.)

Line 35 uses function **`equal_range`** to return a `pair` of forward iterators containing the results of performing both a `lower_bound` and an `upper_bound` operation. The first two arguments must be at least forward iterators. The third is the value for which to locate the equal range. The function returns a `pair` of forward iterators for the lower bound (`eq.first`) and upper bound (`eq.second`), respectively.

Functions `lower_bound`, `upper_bound` and `equal_range` are often used to locate insertion points in sorted sequences. Line 44 uses `lower_bound` to locate the first point at which `5` can be inserted in order in `v`. Line 51 uses `upper_bound` to locate the last point at which `7` can be inserted in order in `v`. Line 59 uses `equal_range` to locate the first and last points at which `5` can be inserted in order in `v`.

22.5.12 Heapsort

Figure 22.37 demonstrates the Standard Library functions for performing the **heapsort sorting algorithm**. Heapsort is a sorting algorithm in which an array of elements is arranged into a special binary tree called a heap. The key features of a heap are that the largest element is always at the top of the heap and the values of the children of any node in the binary tree are always less than or equal to that node's value. A heap arranged in this manner is often called a **maxheap**. Heapsort is discussed in detail in computer science courses called "Data Structures" and "Algorithms."

```
// Fig. 22.37: Fig22_37.cpp
// Standard Library algorithms push_heap, pop_heap,
// make_heap and sort_heap.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   vector< int > v( a, a + SIZE ); // copy of a
   vector< int > v2;
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before make_heap:\n";
   copy( v.begin(), v.end(), output );

   make_heap( v.begin(), v.end() ); // create heap from vector v
   cout << "\nVector v after make_heap:\n";
   copy( v.begin(), v.end(), output );

   sort_heap( v.begin(), v.end() ); // sort elements with sort_heap
   cout << "\nVector v after sort_heap:\n";
   copy( v.begin(), v.end(), output );

   // perform the heapsort with push_heap and pop_heap
   cout << "\n\nArray a contains: ";
   copy( a, a + SIZE, output ); // display array a
   cout << endl;

   // place elements of array a into v2 and
   // maintain elements of v2 in heap
   for ( int i = 0; i < SIZE; i++ )
   {
      v2.push_back( a[ i ] );
      push_heap( v2.begin(), v2.end() );
      cout << "\nv2 after push_heap(a[" << i << "]): ";
      copy( v2.begin(), v2.end(), output );
   } // end for

   cout << endl;

```

Fig. 22.37 | Using Standard Library functions to perform a heapsort. (Part 1 of 2.)

```
46     // remove elements from heap in sorted order
47     for ( unsigned int j = 0; j < v2.size(); j++ )
48     {
49        cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
50        pop_heap( v2.begin(), v2.end() - j );
51        copy( v2.begin(), v2.end(), output );
52     } // end for
53
54     cout << endl;
55  } // end main
```

```
Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22

v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100
```

Fig. 22.37 | Using Standard Library functions to perform a heapsort. (Part 2 of 2.)

Line 21 uses function **make_heap** to take a sequence of values in the range from `v.begin()` up to, but not including, `v.end()` and create a heap that can be used to produce a sorted sequence. The two iterator arguments must be random-access iterators, so this function will work only with arrays, vectors and deques. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 25 uses function **sort_heap** to sort a sequence of values in the range from v.begin() up to, but not including, v.end() that are already arranged in a heap. The two iterator arguments must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 39 uses function **push_heap** to add a new value into a heap. We take one element of array a at a time, appendit to the end of vector v2 and perform the push_heap operation. If the appended element is the only element in the vector, the vector is already a heap. Otherwise, function push_heap rearranges the vector elements into a heap. Each time push_heap is called, it assumes that the last element currently in the vector (i.e., the one that is appended before the push_heap function call) is the element being added to the heap and that all other elements in the vector are already arranged as a heap. The two iterator arguments to push_heap must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 50 uses **pop_heap** to remove the top heap element. This function assumes that the elements in the range specified by its two random-access iterator arguments are already a heap. Repeatedly removing the top heap element results in a sorted sequence of values. Function pop_heap swaps the first heap element (v2.begin()) with the last heap element (the element before v2.end() - i), then ensures that the elements up to, but not including, the last element still form a heap. Notice in the output that, after the pop_heap operations, the vector is sorted in ascending order. A second version of this function takes as a third argument a binary predicate function for comparing values.

22.5.13 min and max

Algorithms **min** and **max** determine the minimum and the maximum of two elements, respectively. Figure 22.38 demonstrates min and max for int and char values.

```cpp
// Fig. 22.38: Fig22_38.cpp
// Standard Library algorithms min and max.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
   cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
   cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
   cout << "\nThe minimum of 'G' and 'Z' is: " << min( 'G', 'Z' );
   cout << "\nThe maximum of 'G' and 'Z' is: " << max( 'G', 'Z' );
   cout << endl;
} // end main
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

Fig. 22.38 | Algorithms min and max.

22.5.14 STL Algorithms Not Covered in This Chapter

Figure 22.39 summarizes the STL algorithms that are not covered in this chapter.

Algorithm	Description
`inner_product`	Calculate the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
`adjacent_difference`	Beginning with the second element in a sequence, calculate the difference (using operator –) between the current and previous elements, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
`partial_sum`	Calculate a running total (using operator +) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.
`nth_element`	Use three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this algorithm executes, all elements before the partitioning element are less than that element and all elements after the partitioning element are greater than or equal to that element. A second version of this algorithm takes as a fourth argument a binary comparison function.
`partition`	This algorithm is similar to `nth_element`, but requires less powerful bidirectional iterators, making it more flexible. It requires two bidirectional iterators indicating the range of elements to partition. The third argument is a unary predicate function that helps partition the elements so that all elements for which the predicate is `true` are to the left (toward the beginning of the sequence) of those for which the predicate is `false`. A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns `false`.
`stable_partition`	Similar to `partition` except that this algorithm guarantees that equivalent elements will be maintained in their original order.
`next_permutation`	Next lexicographical permutation of a sequence.
`prev_permutation`	Previous lexicographical permutation of a sequence.
`rotate`	Use three forward iterator arguments to rotate the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 1 of 2.)

Algorithm	Description
`rotate_copy`	This algorithm is identical to `rotate` except that the results are stored in a separate sequence indicated by the fourth argument—an output iterator. The two sequences must have the same number of elements.
`adjacent_find`	This algorithm returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the `end` of the sequence.
`search`	This algorithm searches for a subsequence of elements within a sequence of elements and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`search_n`	This algorithm searches a sequence of elements looking for a subsequence in which the values of a specified number of elements have a particular value and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`partial_sort`	Use three random-access iterators to sort part of a sequence. The first and last arguments indicate the sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The elements from the second argument iterator to the end of the sequence are in an undefined order.
`partial_sort_copy`	Use two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
`stable_sort`	The algorithm is similar to `sort` except that all equivalent elements are maintained in their original order. This sort is $O(n \log n)$ if enough memory is available; otherwise, it's $O(n(\log n)^2)$.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 2 of 2.)

22.6 Class bitset

Class **`bitset`** makes it easy to create and manipulate **bit sets**, which are useful for representing a set of bit flags. `bitset`s are fixed in size at compile time. Class `bitset` is an alternate tool for bit manipulation, discussed in Chapter 21. The declaration

```
bitset< size > b;
```

creates `bitset` `b`, in which every bit is initially `0`. The statement

```
b.set( bitNumber );
```

sets bit `bitNumber` of `bitset` `b` "on." The expression `b.set()` sets all bits in `b` "on."

The statement

```
b.reset( bitNumber );
```

sets bit `bitNumber` of `bitset` `b` "off." The expression `b.reset()` sets all bits in `b` "off." The statement

```
b.flip( bitNumber );
```

"flips" bit `bitNumber` of `bitset` `b` (e.g., if the bit is on, `flip` sets it off). The expression `b.flip()` flips all bits in `b`. The statement

```
b[ bitNumber ];
```

returns a reference to the bit `bitNumber` of `bitset` `b`. Similarly,

```
b.at( bitNumber );
```

performs range checking on `bitNumber` first. Then, if `bitNumber` is in range, `at` returns a reference to the bit. Otherwise, `at` throws an `out_of_range` exception. The statement

```
b.test( bitNumber );
```

performs range checking on `bitNumber` first. If `bitNumber` is in range, `test` returns `true` if the bit is on, `false` it's off. Otherwise, `test` throws an `out_of_range` exception. The expression

```
b.size()
```

returns the number of bits in `bitset` `b`. The expression

```
b.count()
```

returns the number of bits that are set in `bitset` `b`. The expression

```
b.any()
```

returns `true` if any bit is set in `bitset` `b`. The expression

```
b.none()
```

returns `true` if none of the bits is set in `bitset` `b`. The expressions

```
b == b1
b != b1
```

compare the two `bitset`s for equality and inequality, respectively.

Each of the bitwise assignment operators `&=`, `|=` and `^=` can be used to combine `bitset`s. For example,

```
b &= b1;
```

performs a bit-by-bit logical AND between `bitset`s `b` and `b1`. The result is stored in `b`. Bitwise logical OR and bitwise logical XOR are performed by

```
b |= b1;
b ^= b2;
```

The expression

```
b >>= n;
```

shifts the bits in `bitset` `b` right by `n` positions. The expression

```
b <<= n;
```

shifts the bits in `bitset` `b` left by `n` positions. The expressions

```
b.to_string()
b.to_ulong()
```

convert `bitset` `b` to a `string` and an `unsigned long`, respectively.

Sieve of Eratosthenes with `bitset`

Figure 22.40 revisits the Sieve of Eratosthenes for finding prime numbers that we discussed in Exercise 7.29. A `bitset` is used instead of an array to implement the algorithm. The program displays all the prime numbers from 2 to 1023, then allows the user to enter a number to determine whether that number is prime.

```cpp
// Fig. 22.40: Fig22_40.cpp
// Using a bitset to demonstrate the Sieve of Eratosthenes.
#include <iostream>
#include <iomanip>
#include <cmath>
#include <bitset> // bitset class definition
using namespace std;

int main()
{
   const int SIZE = 1024;
   int value;
   bitset< SIZE > sieve; // create bitset of 1024 bits
   sieve.flip(); // flip all bits in bitset sieve
   sieve.reset( 0 ); // reset first bit (number 0)
   sieve.reset( 1 ); // reset second bit (number 1)

   // perform Sieve of Eratosthenes
   int finalBit = sqrt( static_cast< double >( sieve.size() ) ) + 1;

   // determine all prime numbers from 2 to 1024
   for ( int i = 2; i < finalBit; i++ )
   {
      if ( sieve.test( i ) ) // bit i is on
      {
```

Fig. 22.40 | Class `bitset` and the Sieve of Eratosthenes. (Part 1 of 3.)

```
         for ( int j = 2 * i; j < SIZE; j += i )
            sieve.reset( j ); // set bit j off
      } // end if
   } // end for

   cout << "The prime numbers in the range 2 to 1023 are:\n";

   // display prime numbers in range 2-1023
   for ( int k = 2, counter = 1; k < SIZE; k++ )
   {
      if ( sieve.test( k ) ) // bit k is on
      {
         cout << setw( 5 ) << k;

         if ( counter++ % 12 == 0 ) // counter is a multiple of 12
            cout << '\n';
      } // end if
   } // end for

   cout << endl;

   // get value from user to determine whether value is prime
   cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
   cin >> value;

   // determine whether user input is prime
   while ( value != -1 )
   {
      if ( sieve[ value ] ) // prime number
         cout << value << " is a prime number\n";
      else // not a prime number
         cout << value << " is not a prime number\n";

      cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
      cin >> value;
   } // end while
} // end main
```

```
The prime numbers in the range 2 to 1023 are:
    2    3    5    7   11   13   17   19   23   29   31   37
   41   43   47   53   59   61   67   71   73   79   83   89
   97  101  103  107  109  113  127  131  137  139  149  151
  157  163  167  173  179  181  191  193  197  199  211  223
  227  229  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349  353  359
  367  373  379  383  389  397  401  409  419  421  431  433
  439  443  449  457  461  463  467  479  487  491  499  503
  509  521  523  541  547  557  563  569  571  577  587  593
  599  601  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733  739  743
  751  757  761  769  773  787  797  809  811  821  823  827
  829  839  853  857  859  863  877  881  883  887  907  911
  919  929  937  941  947  953  967  971  977  983  991  997
 1009 1013 1019 1021
```

Fig. 22.40 | Class `bitset` and the Sieve of Eratosthenes. (Part 2 of 3.)

```
Enter a value from 2 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1
```

Fig. 22.40 | Class `bitset` and the Sieve of Eratosthenes. (Part 3 of 3.)

Line 13 creates a `bitset` of `size` bits (`size` is 1024 in this example). By default, all the bits in the `bitset` are set "off." Line 14 calls function **`flip`** to set all bits "on." Numbers 0 and 1 are not prime numbers, so lines 15–16 call function **`reset`** to set bits 0 and 1 "off." Lines 22–29 determine all the prime numbers from 2 to 1023. The integer `finalBit` (line 19) is used to determine when the algorithm is complete. The basic algorithm is that a number is prime if it has no divisors other than 1 and itself. Starting with the number 2, we can eliminate all multiples of that number. The number 2 is divisible only by 1 and itself, so it's prime. Therefore, we can eliminate 4, 6, 8 and so on. The number 3 is divisible only by 1 and itself. Therefore, we can eliminate all multiples of 3 (keep in mind that all even numbers have already been eliminated).

22.7 Function Objects

Many STL algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task. For example, the `binary_search` algorithm that we discussed in Section 22.5.6 is overloaded with a version that requires as its fourth parameter a pointer to a function that takes two arguments and returns a `bool` value. The `binary_search` algorithm uses this function to compare the search key to an element in the collection. The function returns `true` if the search key and element being compared are equal; otherwise, the function returns `false`. This enables `binary_search` to search a collection of elements for which the element type does not provide an overloaded equality `==` operator.

STL's designers made the algorithms more flexible by allowing any algorithm that can receive a function pointer to receive an object of a class that overloads the parentheses operator with a function named `operator()`, provided that the overloaded operator meets the requirements of the algorithm—in the case of `binary_search`, it must receive two arguments and return a `bool`. An object of such a class is known as a **function object** and can be used syntactically and semantically like a function or function pointer—the overloaded parentheses operator is invoked by using a function object's name followed by parentheses containing the arguments to the function. Together, function objects and functions used are know as **functors**. Most algorithms can use function objects and functions interchangeably.

Function objects provide several advantages over function pointers. Since function objects are commonly implemented as class templates that are included into each source code file that uses them, the compiler can inline an overloaded `operator()` to improve performance. Also, since they're objects of classes, function objects can have data members that `operator()` can use to perform its task.

Predefined Function Objects of the Standard Template Library

Many predefined function objects can be found in the header **<functional>**. Figure 22.41 lists several of the STL function objects, which are all implemented as class templates. We used the function object `less< T >` in the `set`, `multiset` and `priority_queue` examples, to specify the sorting order for elements in a container.

STL function objects	Type	STL function objects	Type
`divides< T >`	arithmetic	`logical_or< T >`	logical
`equal_to< T >`	relational	`minus< T >`	arithmetic
`greater< T >`	relational	`modulus< T >`	arithmetic
`greater_equal< T >`	relational	`negate< T >`	arithmetic
`less< T >`	relational	`not_equal_to< T >`	relational
`less_equal< T >`	relational	`plus< T >`	arithmetic
`logical_and< T >`	logical	`multiplies< T >`	arithmetic
`logical_not< T >`	logical		

Fig. 22.41 | Function objects in the Standard Library.

Using the STL `accumulate` Algorithm

Figure 22.42 demonstrates the `accumulate` numeric algorithm (discussed in Fig. 22.30) to calculate the sum of the squares of the elements in a `vector`. The fourth argument to `accumulate` is a **binary function object** (that is, a function object for which `operator()` takes two arguments) or a function pointer to a **binary function** (that is, a function that takes two arguments). Function `accumulate` is demonstrated twice—once with a function pointer and once with a function object.

```
// Fig. 22.42: Fig22_42.cpp
// Demonstrating function objects.
#include <iostream>
#include <vector> // vector class-template definition
#include <algorithm> // copy algorithm
#include <numeric> // accumulate algorithm
#include <functional> // binary_function definition
#include <iterator> // ostream_iterator
using namespace std;

// binary function adds square of its second argument and the
// running total in its first argument, then returns the sum
int sumSquares( int total, int value )
{
   return total + value * value;
} // end function sumSquares

```

Fig. 22.42 | Binary function object. (Part 1 of 2.)

```
// binary function class template defines overloaded operator()
// that adds the square of its second argument and running
// total in its first argument, then returns sum
template< typename T >
class SumSquaresClass : public binary_function< T, T, T >
{
public:
   // add square of value to total and return result
   T operator()( const T &total, const T &value )
   {
      return total + value * value;
   } // end function operator()
}; // end class SumSquaresClass

int main()
{
   const int SIZE = 10;
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > integers( array, array + SIZE ); // copy of array
   ostream_iterator< int > output( cout, " " );
   int result;

   cout << "vector integers contains:\n";
   copy( integers.begin(), integers.end(), output );

   // calculate sum of squares of elements of vector integers
   // using binary function sumSquares
   result = accumulate( integers.begin(), integers.end(),
      0, sumSquares );

   cout << "\n\nSum of squares of elements in integers using "
      << "binary\nfunction sumSquares: " << result;

   // calculate sum of squares of elements of vector integers
   // using binary function object
   result = accumulate( integers.begin(), integers.end(),
      0, SumSquaresClass< int >() );

   cout << "\n\nSum of squares of elements in integers using "
      << "binary\nfunction object of type "
      << "SumSquaresClass< int >: " << result << endl;
} // end main
```

```
vector integers contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary
function sumSquares: 385

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385
```

Fig. 22.42 | Binary function object. (Part 2 of 2.)

Lines 13–16 define a function `sumSquares` that squares its second argument `value`, adds that square and its first argument `total` and returns the sum. Function `accumulate` will pass each of the elements of the sequence over which it iterates as the second argument to `sumSquares` in the example. On the first call to `sumSquares`, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`; `0` in this program). All subsequent calls to `sumSquares` receive as the first argument the running sum returned by the previous call to `sumSquares`. When `accumulate` completes, it returns the sum of the squares of all the elements in the sequence.

Lines 21–30 define a class `SumSquaresClass` that inherits from the class template **`binary_function`** (in header file `<functional>`)—an empty base class for creating function objects in which `operator` receives two parameters and returns a value. Class `binary_function` accepts three type parameters that represent the types of the first argument, second argument and return value of `operator`, respectively. In this example, the type of these parameters is `T` (line 22). On the first call to the function object, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`: `0` in this program) and the second argument will be the first element in `vector` `integers`. All subsequent calls to `operator` receive as the first argument the result returned by the previous call to the function object, and the second argument will be the next element in the `vector`. When `accumulate` completes, it returns the sum of the squares of all the elements in the `vector`.

Lines 45–46 call function `accumulate` with a pointer to function `sumSquares` as its last argument. The statement in lines 53–54 calls function `accumulate` with an object of class `SumSquaresClass` as the last argument. The expression `SumSquaresClass< int >()` creates an instance of class `SumSquaresClass` (a function object) that is passed to `accumulate`, which sends the object the message (invokes the function) `operator`. The statement could be written as two separate statements, as follows:

```
SumSquaresClass< int > sumSquaresObject;
result = accumulate( integers.begin(), integers.end(),
   0, sumSquaresObject );
```

The first line defines an object of class `SumSquaresClass`. That object is then passed to function `accumulate`.

22.8 Wrap-Up

In this chapter, we introduced the Standard Template Library and discussed its three key components—containers, iterators and algorithms. You learned the STL sequence containers, `vector`, `deque` and `list`, which represent linear data structures. We discussed associative containers, `set`, `multiset`, `map` and `multimap`, which represent nonlinear data structures. You also saw that the container adapters `stack`, `queue` and `priority_queue` can be used to restrict the operations of the sequence containers for the purpose of implementing the specialized data structures represented by the container adapters. We then demonstrated many of the STL algorithms, including mathematical algorithms, basic searching and sorting algorithms and set operations. You learned the types of iterators each algorithm requires and that each algorithm can be used with any container that supports the minimum iterator functionality the algorithm requires. You also learned class `bitset`, which makes it easy to create and manipulate bit sets as a container. Finally, we introduced function objects that work syntactically and semantically like ordinary functions, but offer advantages such as performance and the ability to store data.

The next chapter discusses the future of C++. A new standard, known as C++0x, will be released in 2010 or 2011. You'll learn about the new libraries and core language features being added to C++. You'll also learn about the Boost Libraries, which many of the libraries being added to C++0x are based on. We'll demonstrate how to use two of the new libraries to work with regular expressions and smart pointers.

22.9 STL Web Resources

Our C++ Resource Center (`www.deitel.com/cplusplus/`) focuses on the enormous amount of free C++ content available online. Start your search here for resources, downloads, tutorials, documentation, books, e-books, journals, articles, blogs, RSS feeds and more that will help you develop C++ applications. The C++ Resource Center includes links to many STL resources and tutorials.

Summary

Section 22.1 Introduction to the Standard Template Library (STL)

- The Standard Template Library defines powerful, template-based, reusable components that implement many common data structures, and algorithms used to process those data structures.
- The STL has three key components—containers, iterators and algorithms.
- The STL containers are data structures capable of storing objects of any data type. There are three styles of container classes—first-class containers, container adapters and near containers.
- STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements or entire containers.

Section 22.1.1 Introduction to Containers

- The containers are divided into sequence containers, associative containers and container adapters.
- The sequence containers represent linear data structures, such as vectors and linked lists.
- Associative containers are nonlinear containers that quickly locate elements stored in them, such as sets of values or key/value pairs.
- Sequence containers and associative containers are collectively referred to as first-class containers.

Section 22.1.2 Introduction to Iterators

- First-class container function `begin` returns an iterator pointing to the first element of a container. Function `end` returns an iterator pointing to the first element past the end of the container (an element that doesn't exist and is typically used in a loop to indicate when to terminate processing of the container's elements).
- An `istream_iterator` is capable of extracting values in a type-safe manner from an input stream. An `ostream_iterator` is capable of inserting values in an output stream.
- Input and output iterators can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time.
- A forward iterator combines the capabilities of input and output iterators.
- A bidirectional iterator has the capabilities of a forward iterator and the ability to move in the backward direction (i.e., from the end of the container toward the beginning).

- A random-access iterator has the capabilities of a bidirectional iterator and the ability to directly access any element of the container.

Section 22.1.3 Introduction to Algorithms

- Containers that support random-access iterators can be used with all algorithms in the STL.

Section 22.2 Sequence Containers

- The STL provides sequence containers `vector`, `list` and `deque`. Class templates `vector` and `deque` both are based on arrays. Class template `list` implements a linked-list data structure.

Section 22.2.1 `vector` Sequence Container

- Function `capacity` returns the number of elements that can be stored in a vector before the vector dynamically resizes itself to accommodate more elements.
- Sequence container function `push_back` adds an element to the end of a container.
- To use the algorithms of the STL, you must include the header file `<algorithm>`.
- Algorithm `copy` copies each element in a container starting with the location specified by the iterator in its first argument and up to—but not including—the location specified by the iterator in its second argument.
- Function `front` returns a reference to the first element in a sequence container. Function `begin` returns an iterator pointing to the beginning of a sequence container.
- Function `back` returns a reference to the last element in a sequence container. Function `end` returns an iterator pointing to the element one past the end of a sequence container.
- Sequence container function `insert` inserts value(s) before the element at a specific location.
- Function `erase` (in all first-class containers) removes specific element(s) from the container.
- Function `empty` (in all containers and adapters) returns `true` if the container is empty.
- Function `clear` (in all first-class containers) empties the container.

Section 22.2.2 `list` Sequence Container

- The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container. Header file `<list>` must be included to use class template `list`.
- The `list` member function `push_front` inserts values at the beginning of a list.
- The `list` member function `sort` arranges the elements in the list in ascending order.
- The `list` member function `splice` removes elements in one `list` and inserts them into another `list` at a specific position.
- The `list` member function `unique` removes duplicate elements in a `list`.
- The `list` member function `assign` replaces the contents of one `list` with the contents of another.
- The `list` member function `remove` deletes all copies of a specified value from a `list`.

Section 22.2.3 `deque` Sequence Container

- Class template `deque` provides the same operations as `vector`, but adds member functions `push_front` and `pop_front` to allow insertion and deletion at the beginning of a `deque`, respectively. Header file `<deque>` must be included to use class template `deque`.

Section 22.3 Associative Containers

- The STL's associative containers provide direct access to store and retrieve elements via keys.
- The four associative containers are `multiset`, `set`, `multimap` and `map`.

- Class templates `multiset` and `set` provide operations for manipulating sets of values where the values are the keys—there is not a separate value associated with each key. Header file `<set>` must be included to use class templates `set` and `multiset`.
- A `multiset` allows duplicate keys and a `set` does not.

Section 22.3.1 `multiset` Associative Container

- The `multiset` associative container provides fast storage and retrieval of keys and allows duplicate keys. The ordering of the elements is determined by a comparator function object.
- A `multiset`'s keys can be sorted in ascending order by ordering the keys with comparator function object `less<T>`.
- The type of the keys in all associative containers must support comparison properly based on the comparator function object specified.
- A `multiset` supports bidirectional iterators.
- Header file `<set>` must be included to use class `multiset`.

Section 22.3.2 `set` Associative Container

- The `set` associative container is used for fast storage and retrieval of unique keys.
- If an attempt is made to insert a duplicate key into a `set`, the duplicate is ignored.
- A `set` supports bidirectional iterators.
- Header file `<set>` must be included to use class set.

Section 22.3.3 `multimap` Associative Container

- Containers `multimap` and `map` provide operations for manipulating values associated with keys.
- The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.
- Function `count` (available to all associative containers) counts the number of occurrences of the specified value currently in a container.
- Function `find` (available to all associative containers) locates a specified value in a container.
- Functions `lower_bound` and `upper_bound` (available in all associative containers) locate the earliest occurrence of the specified value in a container and the element after the last occurrence of the specified value in a container, respectively.
- Function `equal_range` (available in all associative containers) returns a `pair` containing the results of both a `lower_bound` and an `upper_bound` operation.
- The `multimap` associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs).
- Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is called a one-to-many relationship.
- Header file `<map>` must be included to use class templates `map` and `multimap`.

Section 22.3.4 `map` Associative Container

- Duplicate keys are not allowed in a `map`, so only a single value can be associated with each key. This is called a one-to-one mapping.
- A `map` is commonly called an associative array.

Section 22.4 Container Adapters

- The STL provides three container adapters—`stack`, `queue` and `priority_queue`.

- Adapters are not first-class containers, because they do not provide the actual data structure implementation in which elements can be stored and they do not support iterators.
- All three adapter class templates provide member functions `push` and `pop` that properly insert an element into and remove an element from each adapter data structure, respectively.

Section 22.4.1 `stack` *Adapter*

- Class template `stack` is a last-in, first-out data structure. Header file `<stack>` must be included to use class template `stack`.
- The `stack` member function `top` returns a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container).
- The `stack` member function `empty` determines whether the `stack` is empty (implemented by calling function `empty` of the underlying container).
- The `stack` member function `size` returns the number of elements in the `stack` (implemented by calling function `size` of the underlying container).

Section 22.4.2 `queue` *Adapter*

- Class template `queue` enables insertions at the back of the underlying data structure and deletions from the front of the underlying data structure (commonly referred to as a first-in, first-out data structure). Header file `<queue>` must be included to use a `queue` or a `priority_queue`.
- The `queue` member function `front` returns a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container).
- The `queue` member function `back` returns a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container).
- The `queue` member function `empty` determines whether the `queue` is empty (implemented by calling function `empty` of the underlying container).
- The `queue` member function `size` returns the number of elements in the `queue` (implemented by calling function `size` of the underlying container).

Section 22.4.3 `priority_queue` *Adapter*

- Class template `priority_queue` provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure.
- The common `priority_queue` operations are `push`, `pop`, `top`, `empty` and `size`.

Section 22.5.1 `fill`*,* `fill_n`*,* `generate` *and* `generate_n`

- Algorithms `fill` and `fill_n` set every element in a range of container elements to a specific value.
- Algorithms `generate` and `generate_n` use a generator function or function object to create values for every element in a range of container elements.

Section 22.5.2 `equal`*,* `mismatch` *and* `lexicographical_compare`

- Algorithm `equal` compares two sequences of values for equality.
- Algorithm `mismatch` compares two sequences of values and returns a pair of iterators indicating the location in each sequence of the mismatched elements.
- Algorithm `lexicographical_compare` compares the contents of two sequences.

Section 22.5.3 `remove`*,* `remove_if`*,* `remove_copy` *and* `remove_copy_if`

- Algorithm `remove` eliminates all elements with a specific value in a certain range.
- Algorithm `remove_copy` copies all elements that do not have a specific value in a certain range.

- Algorithm `remove_if` deletes all elements that satisfy the `if` condition in a certain range.
- Algorithm `remove_copy_if` copies all elements that satisfy the `if` condition in a certain range.

Section 22.5.4 `replace`, `replace_if`, `replace_copy` and `replace_copy_if`

- Algorithm `replace` replaces all elements with a specific value in certain range.
- Algorithm `replace_copy` copies all elements with a specific value in a certain range.
- Algorithm `replace_if` replaces all elements that satisfy the `if` condition in a certain range.
- Algorithm `replace_copy_if` copies all elements that satisfy the `if` condition in a certain range.

Section 22.5.5 Mathematical Algorithms

- Algorithm `random_shuffle` reorders randomly the elements in a certain range.
- Algorithm `count` counts the elements with a specific value in a certain range.
- Algorithm `count_if` counts the elements that satisfy the `if` condition in a certain range.
- Algorithm `min_element` locates the smallest element in a certain range.
- Algorithm `max_element` locates the largest element in a certain range.
- Algorithm `accumulate` sums the values in a certain range.
- Algorithm `for_each` applies a general function or function object to every element in a range.
- Algorithm `transform` applies a general function or function object to every element in a range and replaces each element with the result of the function.

Section 22.5.6 Basic Searching and Sorting Algorithms

- Algorithm `find` locates a specific value in a certain range.
- Algorithm `find_if` locates the first value in a certain range that satisfies the `if` condition.
- Algorithm `sort` arranges the elements in a certain range in ascending order or an order specified by a predicate.
- Algorithm `binary_search` determines whether a specific value is in a sorted range of elements.

Section 22.5.7 `swap`, `iter_swap` and `swap_ranges`

- Algorithm `swap` exchanges two values.
- Algorithm `iter_swap` exchanges the two elements.
- Algorithm `swap_ranges` exchanges the elements in a certain range.

Section 22.5.8 `copy_backward`, `merge`, `unique` and `reverse`

- Algorithm `copy_backward` copies elements in a range and places the elements into a container starting from the end and working toward the front.
- Algorithm `merge` combines two sorted ascending sequences of values into a third sorted ascending sequence.
- Algorithm `unique` removes duplicated elements in a certain range of a sorted sequence.
- Algorithm `reverse` reverses all the elements in a certain range.

Section 22.5.9 `inplace_merge`, `unique_copy` and `reverse_copy`

- Algorithm `inplace_merge` merges two sorted sequences of elements in the same container.
- Algorithm `unique_copy` makes a copy of all the unique elements in the sorted sequence of values in a certain range.
- Algorithm `reverse_copy` makes a reversed copy of the elements in a certain range.

Section 22.5.10 Set Operations

- The `set` function `includes` compares two `sets` of sorted values to determine whether every element of the second `set` is in the first `set`.
- The `set` function `set_difference` finds the elements from the first `set` of sorted values that are not in the second `set` of sorted values (both `sets` of values must be in ascending order).
- The `set` function `set_intersection` determines the elements from the first `set` of sorted values that are in the second `set` of sorted values (both `sets` of values must be in ascending order).
- The `set` function `set_symmetric_difference` determines the elements in the first `set` that are not in the second `set` and the elements in the second `set` that are not in the first `set` (both `sets` of values must be in ascending order).
- The `set` function `set_union` creates a `set` of all the elements that are in either or both of the two sorted `sets` (both `sets` of values must be in ascending order).

Section 22.5.11 `lower_bound`, `upper_bound` and `equal_range`

- Algorithm `lower_bound` finds the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.
- Algorithm `upper_bound` finds the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.
- Algorithm `equal_range` performs returns the lower bound and upper bound as a `pair`.

Section 22.5.12 Heapsort

- Algorithm `make_heap` takes a sequence of values in a certain range and creates a heap that can be used to produce a sorted sequence.
- Algorithm `sort_heap` sorts a sequence of values in a certain range of a heap.
- Algorithm `pop_heap` removes the top heap element.

Section 22.5.13 `min` and `max`

- Algorithms `min` and `max` determine the minimum of two elements and the maximum of two elements, respectively.

Section 22.6 Class `bitset`

- Class template `bitset` makes it easy to create and manipulate bit sets, which are useful for representing a set of bit flags.

Section 22.7 Function Objects

- A function object is an instance of a class that overloads `operator()`.
- STL provides many predefined function objects, which can be found in header `<functional>`.
- Binary function objects are function objects that take two arguments and return a value. Class template `binary_function` is an empty base class for creating binary function objects that provides standard type names for the function's parameters and result.

Terminology

`<algorithm>` header file 936
`<deque>` header file 942
`<functional>` header file 992
`<list>` header file 938
`<map>` header file 949
`<numeric>` header file 929

Self-Review Exercises

State whether the following are *true* or *false* or fill in the blanks. If the answer is *false*, explain why,.

22.1 (T/F) The STL makes abundant use of inheritance and virtual functions.

22.2 The two types of first-class STL containers are sequence containers and ________ containers.

22.3 The five main iterator types are ________, ________, ________, ________ and ________.

22.4 (T/F) An iterator acts like a pointer to an element.

22.5 (T/F) STL algorithms can operate on C-like pointer-based arrays.

22.6 (T/F) STL algorithms are encapsulated as member functions within each container class.

22.7 (T/F) When using the remove algorithm on a vector, the algorithm does not decrease the size of the vector from which elements are being removed.

22.8 The three STL container adapters are ________, ________ and ________.

22.9 (T/F) Container member function `end` yields the position of the container's last element.

22.10 STL algorithms operate on container elements indirectly, using ________.

22.11 The `sort` algorithm requires a(n) ________ iterator.

Answers to Self-Review Exercises

22.1 False. These were avoided for performance reasons.

22.2 Associative.

22.3 Input, output, forward, bidirectional, random access.

22.4 True.

22.5 True.

22.6 False. STL algorithms are not member functions. They operate indirectly on containers, through iterators.

22.7 True.

22.8 `stack`, `queue`, `priority_queue`.

22.9 False. It actually yields the position just after the end of the container.

22.10 Iterators.

22.11 Random-access.

Exercises

22.12 *(Palindromes)* Write a function template `palindrome` that takes a `vector` parameter and returns `true` or `false` according to whether the `vector` does or does not read the same forward as backward (e.g., a `vector` containing 1, 2, 3, 2, 1 is a palindrome, but a `vector` containing 1, 2, 3, 4 is not).

22.13 *(Sieve of Eratosthenes)* Modify Fig. 22.40, the Sieve of Eratosthenes, so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number. Remember that a prime number's factors are only 1 and the prime number itself. Every nonprime number has a unique prime factorization. For example, the factors of 54 are 2, 3, 3 and 3. When these values are multiplied together, the result is 54. For the number 54, the prime factors output should be 2 and 3.

22.14 *(Prime Numbers)* Modify Exercise 22.13 so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number and the number of times each prime factor appears in the unique prime factorization. For example, the output for the number 54 should be

```
The unique prime factorization of 54 is: 2 * 3 * 3 * 3
```

Recommended Reading

Ammeraal, L. *STL for C++ Programmers*. New York: John Wiley & Sons, 1997.

Austern, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Boston, MA: Addison-Wesley, 1998.

Glass, G., and B. Schuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

Henricson, M., and E. Nyquist. *Industrial Strength C++: Rules and Recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.

Josuttis, N. *The C++ Standard Library: A Tutorial and Handbook*. Boston, MA: Addison-Wesley, 1999.

Koenig, A., and B. Moo. *Ruminations on C++.* Boston, MA: Addison-Wesley, 1997.

Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Boston, MA: Addison-Wesley, 2001.

Musser, D. R., G. Derge and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition.* Boston, MA: Addison-Wesley, 2002.

Musser, D. R., and A. A. Stepanov. "Algorithm-Oriented Generic Libraries," *Software Practice and Experience,* Vol. 24, No. 7, July 1994.

Nelson, M. *C++ Programmer's Guide to the Standard Template Library.* Foster City, CA: Programmer's Press, 1995.

Pohl, I. *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide.* Boston, MA: Addison-Wesley, 1997.

Pohl, I. *Object-Oriented Programming Using C++, Second Edition*. Boston, MA: Addison-Wesley, 1997.

Reese, G. *C++ Standard Library Practical Tips.* Hingham, MA: Charles River Media, 2005.

Robson, R. *Using the STL: The C++ Standard Template Library, Second Edition*. New York: Springer, 2010.

Schildt, H. *STL Programming from the Ground Up*, New York: Osborne McGraw-Hill, 1999.

Stepanov, A., and M. Lee. "The Standard Template Library," *Internet Distribution* 31 October 1995 `<www.cs.rpi.edu/~musser/doc.ps>`.

Stroustrup, B. "Making a `vector` Fit for a Standard," *The C++ Report,* October 1994.

Stroustrup, B. *The Design and Evolution of C++*. Boston, MA: Addison-Wesley, 1994.

Stroustrup, B. *The C++ Programming Language, Third Edition.* Boston, MA: Addison-Wesley, 1997.

Vandevoorde, D., and N. Josuttis. *C++ Templates: The Complete Guide.* Boston, MA: Addison-Wesley, 2002.

Vilot, M. J. "An Introduction to the Standard Template Library," *The C++ Report,* Vol. 6, No. 8, October 1994.

Wilson, M. *Extended STL, Volume 1: Collections and Iterators.* Boston, MA: Addison-Wesley, 2007