**CHAPTER 23**

**STL Algorithms**

Objectives

- To use various types of iterators with the STL algorithms (§§23.1–23.20).
- To discover the four types of STL algorithms: nonmodifying algorithms, modifying algorithms, numeric algorithms, and heap algorithms (§23.2).
- To use the `copy` algorithm to copy contents (§23.3).
- To use the algorithms `fill`, `fill_n` to fill values (§23.4).
- To pass functions as parameters (§23.5).
- To use the algorithms `generate`, and `generate_n` (§23.6).
- To use the algorithms `remove`, `remove_if`, `remove_copy`, and `remove_copy_if` (§23.7).
- To use Boolean functions to specify criteria for STL algorithms (§23.8).
- To use the algorithms `replace`, `replace_if`, `replace_copy`, and `replace_copy_if`, `find`, `find_if`, `find_end`, and `find_first_of` (§§23.8–23.9).
- To use the algorithms `search`, `search_n`, `sort`, `binary_search`, `adjacent_find`, `merge`, and `inplace_merge` (§§23.10–23.12).
- To use function objects in STL algorithms (§23.11).
- To use the algorithms `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `swap`, `iter_swap`, and `swap_range` (§§23.13–23.15).
- To use the algorithms `count`, `count_if`, `max_element`, `min_element`, `random_shuffle`, `for_each`, and `transform` (§§23.16–23.19).
- To use set algorithms `includes`, `set_union`, `set_difference`, `set_intersection`, and `set_symmetric_difference` (§23.20).
- To use numeric algorithms `accumulate`, `adjacent_difference`, `inner_product`, and `partial_sum` (§23.21).

### 23.1 Introduction

Key Point: *STL provides algorithms for manipulating the elements in the containers through iterators.*

Often you need to find an element in a container, replace an element with a new element in a container, remove elements from a container, fill the container with some elements, reverse the elements in a container, find the maximum or minimum elements in a container, and sort the elements in a container. These functions are common to all containers. Rather than implementing them in each container, the STL supports them as generic algorithms that can be applied to a variety of containers as well as arrays. The algorithms operate on the elements through iterators. For example, you can apply the STL algorithm **`max_element`** in a vector and in a set using iterators as follows:

```
vector<int> v; v.push_back(4); v.push_back(14); v.push_back(1);
set<int> s; s.insert(4); s.insert(14); s.insert(1);

cout << "The max element in the vector is " <<
  *max_element(v.begin(), v.end()) << endl;

cout << "The max element in the set is " <<
  *max_element(s.begin(), s.end()) << endl;
```

Prior to the STL, algorithms were implemented in the classes with inheritance and polymorphism. The STL separates algorithms from the containers. This enables the algorithms to be applied generically to all containers through iterators. The STL makes the algorithms and containers easy to maintain.

> NOTE:
>
> The terms operations, algorithms, and functions are interchangeable. Functions are operations, and functions are implemented using algorithms.

> NOTE:
>
> Iterators are a generalization of pointers. Pointers themselves are iterators. So, the array pointers can be treated as iterators. Iterators are often used with containers, but some iterators such as **`istream_iterator`** and **`ostream_iterator`** are not associated with containers.

### 23.2 Types of Algorithms

Key Point: STL algorithms can be classified into nonmodifying algorithms, modifying algorithms, numeric algoriuthms, and heap algorithms.

The STL provides approximately 80 algorithms. They can be classified into four groups:

- ***Nonmodifying Algorithms***: Nonmodifying algorithms do not change the contents in the container. They obtain information from the elements. The nonmodifying algorithms are listed in Table 23.1.
- ***Modifying Algorithms***: Modifying algorithms modify the elements in the container by insertion, removing, rearranging, and changing the values of the elements. Table 23.2 lists these algorithms.
- ***Numeric Algorithms***: Numeric algorithms provide four numeric operations for computing accumulate, adjacent difference, partial sum, and inner product. Table 23.3 lists these algorithms.
- ***Heap Algorithms***: Heap algorithms provide four operations for creating a heap, removing and inserting elements from/to a heap, and sorting a heap.

Table 23.1

Nonmodifying Algorithms

| | | | |
|---|---|---|---|
| `adjacent_find` | `find` | `lower_bound` | `search` |
| `binary_search` | `find_end` | `mismatch` | `search_n` |
| `count` | `find_first_of` | `max` | `upper_bound` |
| `count_if` | `find_if` | `max_element` | |
| `equal` | `for_each` | `min` | |
| `equal_range` | `includes` | `min_element` | |

Table 23.2

Modifying Algorithms

| | | |
|---|---|---|
| copy | prev_permutation | rotate_copy |
| copy_backward | random_shuffle | set_difference |
| fill | remove | set_intersection |
| fill_n | remove_copy | set_symmetric_difference |
| generate | remove_copy_if | set_union |
| generate_n | remove_if | sort |
| inplace_merge | replace | stable_partition |
| iter_swap | replace_copy | stable_sort |
| merge | replace_copy_if | swap |
| next_permutation | replace_if | swap_ranges |
| nth_element | reverse | transform |
| partial_sort | reverse_copy | unique |
| partial_sort_copy | rotate | unique_copy |
| partition | | |

Table 23.3

Numeric Algorithms

| | | | |
|---|---|---|---|
| accumulate | adjacent_difference | inner_product | partial_sum |

Table 23.4

Heap Algorithms

| | | | |
|---|---|---|---|
| make_heap | pop_heap | push_heap | sort_heap |

The numeric algorithms are contained in the **<numeric>** header file, and all the other algorithms are contained in the **<algorithm>** header file.

All the algorithms operate through iterators. Recall that the STL defines five types of iterators: input, output, forward, bidirectional, and random-access. The containers **vector** and **deque** support random-access iterators, and **list**, **set**, **multiset**, **map**, and **multimap** support bidirectional iterators. Most of the algorithms require a forward iterator. If an algorithm works with a weak iterator, it can automatically work with a stronger iterator.

Many algorithms operate on a sequence of elements pointed by two iterators. The first iterator points to the first element of the sequence, and the second points to the element after the last element of the sequence.

The rest of the chapter gives examples to demonstrate some frequently used algorithms.

### *Check point*

23.1 Are the STL algorithms defined in a container class such as **`vector`**, **`list`**, or **`set`**? Which header file defines the STL algorithms?

23.2 What are the four types of STL algorithms?

## 23.3 `copy`

Key Point: The **`copy`** function can be used to copy elements in a sequence from one container to another.

The syntax is

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator beg, InputIterator end,
  OutputIterator targetPosition)
```

The function copies the elements within the ranges **`beg .. end – 1`** from a source container to a target container starting at **`targetPosition`**, where **`beg`** and **`end`** are the iterators in the source container and **`targetPosition`** is the iterator in the target container. The function returns an iterator that points to the next position past the last element copied.

Listing 23.1 demonstrates how to use the **`copy`** function.

**Listing 23.1 CopyDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <list>
 5  #include <iterator>
 6  using namespace std;
 7
 8  int main()
 9  {
10    int values[] = {1, 2, 3, 4, 5, 6};
```

```
11    vector<int> intVector(5);
12    list<int> intList(5);
13
14    copy(values + 2, values + 4, intVector.begin());
15    copy(values, values + 5, intList.begin());
16
17    cout << "After initial copy intVector: ";
18    for (int& e: intVector)
19      cout << e << " ";
20
21    cout << "\nAfter initial copy intList: ";
22    for (int& e: intList)
23      cout << e << " ";
24
25    intVector.insert(intVector.begin(), 747);
26    ostream_iterator<int> output(cout, " ");
27    cout << "\nAfter the insertion function, intVector: ";
28    copy(intVector.begin(), intVector.end(), output);
29
30    cout << "\nAfter the copy function, intList: ";
31    copy(intVector.begin(), intVector.begin() + 4, intList.begin());
32    copy(intVector.begin(), intVector.end(), output);
33
34    return 0;
35  }
```

***Sample output***

```
After initial copy intVector: 3 4 0 0 0
After initial copy intList: 1 2 3 4 5
After the insertion function, intVector: 747 3 4 0 0 0
After the copy function, intList: 747 3 4 0 5
```

The program creates an array (line 10), a vector (line 11), and a list (line 12). The **`copy`** function copies the elements **`values[2]`** and **`values[3]`** to the beginning of the vector (line 14), and the elements **`values[0]`**, **`values[1]`**, **`values[2]`**, **`values[3]`**, and **`values[4]`** to the beginning of the list (line 15).

You can copy elements from an array to a container. You can also copy elements from a container to an array or to an output stream.

The program inserts a new element to the vector (line 30), creates an output stream iterator (line 31), and copies the vector to the output stream iterator (line 33):

```
copy(intVector.begin(), intVector.end(), output);
```

The elements in the list are displayed similarly (line 37).

TIP:

**ostream_iterator** was introduced in §22.3.4. It is convenient to use the **copy** function to write the elements from a container to an output stream.

CAUTION:

Before copying **n** elements from a source to a target, the elements in the target must already exist. Otherwise, a runtime error may occur. For example, the following code will cause a runtime error, because the vector is empty:

```
int values[] = {1, 2, 3, 4, 5, 6};
vector<int> intVector;
copy(values + 2, values + 4, intVector.begin()); // Error
```

***Check point***

23.3 What is the **copy** algorithm for? What is the return value of the **copy** algorithm? Show the printout of the following code:

```
int values[] = {1, 2, 3, 4, 5};
vector<int> intVector(5);

vector<int>::iterator last =
  copy(values, values + 3, intVector.begin());

ostream_iterator<int> output(cout, " ");
cout << "intVector: ";
copy(intVector.begin(), last, output);
```

23.4 What is wrong in the following code?

```
int values[] = {1, 2, 3, 4, 5, 6, 7};
vector<int> intVector(5);

vector<int>::iterator last =
  copy(values, values + 7, intVector.begin());
```

**23.4 fill and fill_n**

Key Point: The **fill** and **fill_n** functions can be used to fill a container with a specified value.

The syntax for the **fill** function is as follows:

```
template<typename ForwardIterator, typename T>
void fill(ForwardIterator beg, ForwardIterator end, const T& value)
```

The **fill_n** function can be used to fill a container with a specified value for the elements from iterator **beg** to **beg + n - 1**, using the following syntax:

```
template<typename ForwardIterator, typename size, typename T>
void fill_n(ForwardIterator beg, size n, const T& value)
```

Listing 23.2 demonstrates how to use these two functions.

**Listing 23.2 FillDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int values[] = {1, 2, 3, 4, 5, 6};
10    list<int> intList(values, values + 6);
11
12    ostream_iterator<int> output(cout, " ");
13    cout << "Initial contents, values: ";
14    copy(values, values + 6, output);
15    cout << "\nInitial contents, intList: ";
16    copy(intList.begin(), intList.end(), output);
17
18    fill(values + 2, values + 4, 88);
19    fill_n(intList.begin(), 2, 99);
20
21    cout << "\nAfter the fill function, values: ";
22    copy(values, values + 6, output);
23    cout << "\nAfter the fill_n function, intList: ";
24    copy(intList.begin(), intList.end(), output);
25
26    return 0;
27  }
```

***Sample output***

```
Initial contents, values: 1 2 3 4 5 6
Initial contents, intList: 1 2 3 4 5 6
After the fill function, values: 1 2 88 88 5 6
After the fill_n function, intList: 99 99 3 4 5 6
```

The program creates an array (line 9) and a list (line 10). The **fill** function (line 18) fills **88** to the array in **values[2]** and **values[3]**. The **fill_n** function (line 19) fills 2 elements with value **99** starting from the beginning of the list.

### *Check point*

23.5 What are the **fill** and **fill_n** algorithms for? Show the printout of the following code:

```
int values[] = {1, 2, 3, 4, 5};

fill_n(values + 2, 2, 9);


ostream_iterator<int> output(cout, " ");

cout << "values: ";

copy(values, values + 5, output);
```

### 23.5 Passing Functions as Parameters

Key Point: In C++, you can pass functions as parameters in a function.

Many STL functions contain parameters that are functions. To pass a function parameter is to pass the address of the function. We demonstrate this capability using three examples.

The first example, given in Listing 23.3, defines a function with a function parameter that has its own parameters.

**Listing 23.3 FunctionWithFunctionParameter1.cpp**

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int f1(int value)
 5  {
 6    return 2 * value;
 7  }
 8
 9  int f2(int value)
10  {
11    return 3 * value;
12  }
13
14  void m(int t[], int size, int f(int))
15  {
16    for (int i = 0; i < size; i++)
17      t[i] = f(t[i]);
18  }
19
20  int main()
21  {
22    int list1[] = {1, 2, 3, 4};
23    m(list1, 4, f1);
```

```
24    for (int i = 0; i < 4; i++)
25      cout << list1[i] << " ";
26    cout << endl;
27
28    int list2[] = {1, 2, 3, 4};
29    m(list2, 4, f2);
30    for (int i = 0; i < 4; i++)
31      cout << list2[i] << " ";
32      cout << endl;
33
34    return 0;
35  }
```

***Sample output***

```
2 4 6 8
3 6 9 12
```

Function **m** takes three parameters. The last parameter is a function that takes an **int** parameter and returns and **int** value.

The statement

```
m(list1, 4, f1);
```

invokes function **m** with three arguments. The last argument is function **f1**.

The statement

```
m(list2, 4, f2);
```

invokes function **m** with three arguments. The last argument is function **f2**.

Note that line 14

```
void m(int t[], int size, int f(int))
```

is same as

```
void m(int t[], int size, int (*f)(int))
```

Line 17 can be written as **t[i] = (*f)(t[i])** and line 23 can be written as **m(list1, 4, *f1)**.

The second example, given in Listing 23.4, defines a function with a function parameter that has no parameters.

**Listing 23.4 FunctionWithFunctionParameter2.cpp**

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int nextNum()
 5  {
 6    static int n = 20;
 7    return n++;
 8  }
 9
10  void m(int t[], int size, int f())
11  {
12    for (int i = 0; i < size; i++)
13      t[i] = f();
14  }
15
16  int main()
17  {
18    int list[4];
19    m(list, 4, nextNum);
20    for (int i = 0; i < 4; i++)
21      cout << list[i] << " ";
22    cout << endl;
23
24    return 0;
25  }
```

***Sample output***

```
20 21 22 23
```

Function `m` takes three parameters. The last parameter is a function that takes no parameters and returns an `int` value.

The third example in Listing 23.5 defines a function with a function parameter that has no return value.

**Listing 23.5 FunctionWithFunctionParameter3.cpp**

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  void print(int number)
 5  {
 6    cout << 2 * number << " ";
 7  }
 8
 9  void m(int t[], int size, void f(int))
10  {
11    for (int i = 0; i < size; i++)
12      f(t[i]);
13  }
14
15  int main()
16  {
```

```
17    int list[4] = {1, 2, 3, 4};
18    m(list, 4, print);
19
20    return 0;
21  }
```

***Sample output***

```
2 4 6 8
```

Function **m** takes three parameters. The last parameter is a function that does not have a return value.

NOTE:

You can also pass the function address to a pointer. For example, the following statement passes the **f1** function in Listing 23.3 to a pointer **p**.

```
int (*p)(int) = f1;
```

The following statement passes the **nextNum** function in Listing 23.4 to a pointer **p**.

```
int (*p)(int) = nextNum;
```

The following statement passes the **print** function in Listing 23.5 to a pointer **p**.

```
void (*p)(int) = print;
```

## 23.6 generate and generate_n

Key Point: *The functions **generate** and **generate_n** fill a sequence with a value returned from a function.*

The syntax for these functions are as follows:

```
template<typename ForwardIterator, typename function>
void generate(ForwardIterator beg, ForwardIterator end, function gen)

template<typename ForwardIterator, typename size, typename function>
void generate_n(ForwardIterator beg, size n, function gen)
```

Listing 23.6 demonstrates how to use these two functions.

**Listing 23.6 GenerateDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
3  #include <list>
4  #include <iterator>
5  using namespace std;
6
7  int nextNum()
```

```
 8  {
 9    static int n = 20;
10    return n++;
11  }
12
13  int main()
14  {
15    int values[] = {1, 2, 3, 4, 5, 6};
16    list<int> intList(values, values + 6);
17
18    ostream_iterator<int> output(cout, " ");
19    cout << "Initial contents, values: ";
20    copy(values, values + 6, output);
21    cout << "\nInitial contents, intList: ";
22    copy(intList.begin(), intList.end(), output);
23
24    generate(values + 2, values + 4, nextNum);
25    generate_n(intList.begin(), 2, nextNum);
26
27    cout << "\nAfter the fill function, values: ";
28    copy(values, values + 6, output);
29    cout << "\nAfter the fill_n function, intList: ";
30    copy(intList.begin(), intList.end(), output);
31
32    return 0;
33  }
```

***Sample output***

```
Initial contents, values: 1 2 3 4 5 6
Initial contents, intList: 1 2 3 4 5 6
After the generate function, values: 1 2 20 21 5 6
After the generate_n function, intList: 22 23 3 4 5 6
```

The program creates an array (line 15) and a list (line 16). The **`generate`** function (line 24) fills the array elements **`values[2]`** and **`values[3]`** with the values generated from the **`nextNum`** function. Note that **`n`** is a static local variable (line 9), so its value is persistent for the lifetime of the program. Invoking **`nextNum()`** returns **`20`** the first time, **`21`** the second time, and the return value is always one more for the next call.

***Check point***

23.6 What are the **`generate`** and **`generate_n`** algorithms for? Show the printout of the following code:

```
int nextNum()
{
  static int n = 20;
  return n++;
}

int main()
{
```

```
  int values[] = {1, 2, 3, 4, 5};
  generate_n(values + 1, 2, nextNum);

  ostream_iterator<int> output(cout, " ");
  cout << "values: ";
  copy(values, values + 5, output);

  return 0;
}
```

## 23.7 remove, remove_if, remove_copy, and remove_copy_if

Key Point: STL provides four functions for removing elements in a container: **remove**, **remove_if**, **remove_copy**, and **remove_copy_if**.

The function **remove** removes the elements from a sequence that matches the specified **value**, using the following syntax:

```
template<typename ForwardIterator, typename T>
ForwardIterator remove(ForwardIterator beg,
  ForwardIterator end, const T& value)
```

The function **remove_if** removes all the elements from a sequence such that **boolFunction(element)** is **true**, using the following syntax:

```
template<typename ForwardIterator, typename boolFunction>
ForwardIterator remove_if(ForwardIterator beg,
  ForwardIterator end, boolFunction f)
```

Both **remove** and **remove_if** return an iterator that points to the position after the last element of the new range of the elements.

The function **remove_copy** copies all the elements in the sequence to the target container, except those whose value matches the specified value, using the following syntax:

```
template<typename InputIterator, typename OutputIterator, typename T>
OutputIterator remove_copy(InputIterator beg, InputIterator end,
  OutputIterator targetPosition, const T& value)
```

The function **remove_copy_if** copies all the elements in the sequence to the target container, except those for which **boolFunction(element)** is **true**, using the following syntax:

```
template<typename InputIterator, typename OutputIterator,
  typename boolFunction>
OutputIterator remove_copy_if(InputIterator beg, InputIterator end,
```

```
        OutputIterator targetPosition, boolFunction f)
```

Both **remove_copy** and **remove_copy_if** return an iterator that points to the position after the last element copied.

> NOTE:
>
> Some STL algorithms allow you to pass the pointer of a Boolean function. The Boolean function is used to check whether an element satisfies a condition. For example, you may define a function named **greaterThan3(int element)** to check whether an element is greater than **3**.

> NOTE:
>
> These four functions do not change the size of the container. Elements are moved to the beginning of the container. For example, suppose a list contains elements {**1**, **2**, **3**, **4**, **5**, **6**}. After removing 2, the list contains {**1**, **3**, **4**, **5**, **6**, **6**}. Note that the last element is **6**.

> NOTE:
>
> The **remove_copy** and **remove_copy_if** functions copy the new contents to the target container but do not change the source container.

Listing 23.7 demonstrates how to use the functions **remove** and **remove_if**.

**Listing 23.7 RemoveDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  bool greaterThan3(int value)
 8  {
 9    return value > 3;
10  }
11
```

```
12  int main()
13  {
14    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
15    list<int> intList(values, values + 8);
16
17    ostream_iterator<int> output(cout, " ");
18    cout << "Initial contents, values: ";
19    copy(values, values + 8, output);
20    cout << "\nInitial contents, intList: ";
21    copy(intList.begin(), intList.end(), output);
22
23    remove(values, values + 8, 3);
24    remove_if(intList.begin(), intList.end(), greaterThan3);
25
26    cout << "\nAfter the remove function, values: ";
27    copy(values, values + 8, output);
28    cout << "\nAfter the remove_if function, intList: ";
29    copy(intList.begin(), intList.end(), output);
30
31    return 0;
32  }
```

***Sample output***

```
Initial contents, values: 1 7 3 4 3 6 1 2
Initial contents, intList: 1 7 3 4 3 6 1 2
After the remove function, values: 1 7 4 6 1 2 1 2
After the remove_if function, intList: 1 3 3 1 2 6 1 2
```

The program creates an array (line 14), a list (line 15), and displays their initial contents (lines 17–21). The array is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}. The **`remove`** function (line 23) removes **3** from the array. The new array is {**1**, **7**, **4**, **6**, **1**, **2**, **1**, **2**}.

The **`remove_if`** function (line 24) removes all the elements in the list such that **`greaterThan3(element)`** is **`true`**. Before invoking the function, the list is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}. After invoking the function, the list becomes {**1**, **3**, **3**, **1**, **2**, **6**, **1**, **2**}.

Listing 23.8 demonstrates how to use the functions **`remove_copy`** and **`remove_copy_if`**.

**Listing 23.8 RemoveCopyDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
3  #include <list>
4  #include <iterator>
5  using namespace std;
6
```

```
 7  bool greaterThan3(int value)
 8  {
 9    return value > 3;
10  }
11
12  int main()
13  {
14    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
15    list<int> intList(values, values + 8);
16
17    ostream_iterator<int> output(cout, " ");
18    cout << "Initial contents, values: ";
19    copy(values, values + 8, output);
20    cout << "\nInitial contents, intList: ";
21    copy(intList.begin(), intList.end(), output);
22
23    int newValues[] = {9, 9, 9, 9, 9, 9, 9, 9};
24    list<int> newIntList(values, values + 8);
25    remove_copy(values, values + 8, newValues, 3);
26    remove_copy_if(intList.begin(), intList.end(),
newIntList.begin(),
27      greaterThan3);
28
29    cout << "\nAfter the remove_copy function, values: ";
30    copy(values, values + 8, output);
31    cout << "\nAfter the remove_copy_if function, intList: ";
32    copy(intList.begin(), intList.end(), output);
33    cout << "\nAfter the remove_copy function, newValues: ";
34    copy(newValues, newValues + 8, output);
35    cout << "\nAfter the remove_copy_if function, newIntList: ";
36    copy(newIntList.begin(), newIntList.end(), output);
37
38    return 0;
39  }
```

***Sample output***

```
Initial contents, values: 1 7 3 4 3 6 1 2
Initial contents, intList: 1 7 3 4 3 6 1 2
After the remove_copy function, values: 1 7 3 4 3 6 1 2
After the remove_copy_if function, intList: 1 7 3 4 3 6 1 2
After the remove_copy function, newValues: 1 7 4 6 1 2 9 9
After the remove_copy_if function, newIntList: 1 3 3 1 2 6 1 2
```

The **`remove_copy`** function (line 25) removes **`3`** from array **`values`** and copies the rest to array **`newValues`**. The content of the original array is not changed. Before the copy, array **`values`** is {**`1`**, **`7`**, **`3`**, **`4`**, **`3`**, **`6`**, **`1`**, **`2`**} and **`newValues`** is {**`9`**, **`9`**, **`9`**, 9, **`9`**, **`9`**, **`9`**, **`9`**}. After the copy, array **`newValue`**s becomes {**`1`**, **`7`**, **`4`**, **`6`**, **`1`**, **`2`**, **`9`**, **`9`**}.

The **`remove_copy_if`** function (line 26) removes all the elements in the list such that **`greaterThan3(element)`** is **`true`** and copies the rest to list **`newIntList`**. The content of the

original list is not changed. Before the copy, list **intList** is {**1**, **7**, **3**, **4**, **3**, **6**, 1, **2**} and **newIntList** is {**9**, **9**, **9**, **9**, **9**, **9**, **9**, **9**}. After the copy, list **newIntList** becomes {**1**, **3**, **3**, **1**, **2**, **6**, **1**, **2**}.

***Check point***

23.7 What are the **remove**, **remove_if**, **remove_copy**, and **remove_copy_if** algorithms for?

Show the printout of the following code:

```
bool greaterThan4(int value)
{
  return value > 4;
}

int main()
{
  int values[] = {1, 2, 3, 4, 5, 1, 1};
  remove_if(values, values + 7, greaterThan4);

  ostream_iterator<int> output(cout, " ");
  cout << "values: ";
  copy(values, values + 7, output);

  return 0;
}
```

## 23.8 replace, replace_if, replace_copy, and replace_copy_if

Key Point: *STL provides four functions for replacing elements in a container:* ***replace***, ***replace_if***, ***replace_copy***, *and* ***replace_copy_if***.

The **replace** function replaces all occurrences of a given value with a new value in a sequence, using the following syntax:

```
template<typename ForwardIterator, typename T>
void replace(ForwardIterator beg, ForwardIterator end,
  const T& oldValue, const T& newValue)
```

The **replace_if** function replaces all occurrences of the element for which **boolFunction(element)** is **true** with a new value, using the following syntax:

```
template<typename ForwardIterator, typename boolFunction, typename T>
void replace_if(ForwardIterator beg, ForwardIterator end,
  boolFunction f, const T& newValue)
```

The function **replace_copy** replaces the occurrence of a given value with a new value and copies the result to the target container, using the following syntax:

```
template<typename ForwardIterator, typename OutputIterator,
  typename T>
ForwardIterator replace_copy(ForwardIterator beg,
  ForwardIterator end, OutputIterator targetPosition,
  T& oldValue, T& newValue)
```

The function **replace_copy_if** replaces the occurrence of the element for which **boolFunction(element)** is **true** with a new value, and copies all the elements in the sequence to the target container, using the following syntax:

```
template<typename ForwardIterator, typename OutputIterator,
  typename boolFunction, typename T>
ForwardIterator replace_copy_if(ForwardIterator beg,
  ForwardIterator end, OutputIterator targetPosition,
  boolFunction f, const T& newValue)
```

Both **replace_copy** and **replace_copy_if** return an iterator that points to the position after the last element copied.

NOTE:

The **replace_copy** and **replace_copy_if** functions copy the new contents to the target container but do not change the source container.

Listing 23.9 demonstrates how to use the functions **replace** and **replace_if**.

**Listing 23.9 ReplaceDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  bool greaterThan3(int value)
 8  {
 9    return value > 3;
10  }
11
12  int main()
13  {
14    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
15    list<int> intList(values, values + 8);
16
17    ostream_iterator<int> output(cout, " ");
18    cout << "Initial contents, values: ";
```

```
19    copy(values, values + 8, output);
20    cout << "\nInitial contents, intList: ";
21    copy(intList.begin(), intList.end(), output);
22
23    replace(values, values + 8, 3, 747);
24    replace_if(intList.begin(), intList.end(), greaterThan3, 747);
25
26    cout << "\nAfter the replace function, values: ";
27    copy(values, values + 8, output);
28    cout << "\nAfter the replace_if function, intList: ";
29    copy(intList.begin(), intList.end(), output);
30
31    return 0;
32  }
```

***Sample output***

```
Initial contents, values: 1 7 3 4 3 6 1 2
Initial contents, intList: 1 7 3 4 3 6 1 2
After the replace function, values: 1 7 747 4 747 6 1 2
After the replace_if function, intList: 1 747 3 747 3 747 1 2
```

The program creates an array (line 14), a list (line 15), and displays their initial contents (lines 17–21). The array is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}. The **replace** function (line 23) replaces **3** with **747** in the array. The new array is {**1**, **7**, **747**, **4**, **747**, **6**, **1**, **2**}

The **replace_if** function (line 24) replaces all the elements in the list such that **greaterThan3(element)** is **true**. Before invoking the function, the list is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}. After invoking the function, the list becomes {**1**, **747**, **3**, **747**, **3**, **747**, **1**, **2**}.

Listing 23.10 demonstrates how to use the functions **replace_copy** and **replace_copy_if**.

**Listing 23.10 ReplaceCopyDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
3  #include <list>
4  #include <iterator>
5  using namespace std;
6
7  bool greaterThan3(int value)
8  {
9    return value > 3;
10 }
11
12 int main()
13 {
14   int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
15   list<int> intList(values, values + 8);
16
17   ostream_iterator<int> output(cout, " ");
18   cout << "Initial contents, values: ";
19   copy(values, values + 8, output);
20   cout << "\nInitial contents, intList: ";
21   copy(intList.begin(), intList.end(), output);
```

```
22
23    int newValues[] = {9, 9, 9, 9, 9, 9, 9, 9};
24    list<int> newIntList(values, values + 8);
25    replace_copy(values + 2, values + 5, newValues, 3, 88);
26    replace_copy_if(intList.begin(), intList.end(),
27      newIntList.begin(), greaterThan3, 88);
28
29    cout << "\nAfter the replace function, values: ";
30    copy(values, values + 8, output);
31    cout << "\nAfter the replace_if function, intList: ";
32    copy(intList.begin(), intList.end(), output);
33    cout << "\nAfter the replace_copy function, newValues: ";
34    copy(newValues, newValues + 8, output);
35    cout << "\nAfter the replace_copy_if function, newIntList: ";
36    copy(newIntList.begin(), newIntList.end(), output);
37
38    return 0;
39  }
```

***Sample output***

```
Initial contents, values: 1 7 3 4 3 6 1 2
Initial contents, intList: 1 7 3 4 3 6 1 2
After the replace_copy function, values: 1 7 3 4 3 6 1 2
After the replace_copy_if function, intList: 1 7 3 4 3 6 1 2
After the replace_copy function, newValues: 88 4 88 9 9 9 9 9
After the replace_copy_if function, newIntList: 1 88 3 88 3 88 1 2
```

The **replace_copy** function (line 25) replaces **3** by **88** in array **values** and copies a partial array to array **newValues**. The content of the original array is not changed. Before the replacement, array **values** is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**} and **newValues** is {**9**, **9**, **9**, **9**, **9**, **9**, **9**, **9**}. After the replacement, array **newValue**s becomes {**88**, **4**, **88**, 9, **9**, **9**, **9**, **9**}. Note that only a partial array from position **2** to **4** is copied to the target starting at position **0**.

The **replace_copy_if** function (line 26) replaces all the elements in the list such that **greaterThan3(element)** is **true**, and copies the rest to list **newIntList**. The content of the original list is not changed. Before the replacement, list **intList** is {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**} and **newIntList** is {**9**, **9**, **9**, **9**, **9**, **9**, **9**, **9**}. After the replacement, list **newIntList** becomes {**1**, **88**, **3**, **88**, **3**, **88**, **1**, **2**}.

***Check point***

23.8 What are the **replace**, **replace_if**, **replace_copy**, and **replace_copy_if** algorithms for? Show the printout of the following code:

```
bool greaterThan4(int value)
{
  return value > 4;
}

int main()
{
  int values[] = {1, 2, 3, 4, 5, 1, 1};
  replace_if(values, values + 7, greaterThan4, 999);

  ostream_iterator<int> output(cout, " ");
  cout << "values: ";
  copy(values, values + 7, output);

  return 0;
}
```

### 23.9 find, find_if, find_end, and find_first_of

Key Point: *STL provides four functions for searching an element in a container:* ***find***, ***find_if***, ***find_end***, *and* ***find_first_of***.

The **find** function searches for an element, using the syntax:

```
template<typename InputIterator, typename T>
InputIterator find(InputIterator beg, InputIterator end, T& value)
```

The **find_if** function searches for an element such that **boolFunction(element)** is **true**, using the syntax:

```
template<typename InputIterator, typename boolFunction>
InputIterator find_if(InputIterator beg, InputIterator end,
  boolFunction f)
```

Both functions return the iterator that points to the first matching element if found; otherwise, return **end**.

Listing 23.11 demonstrates how to use the functions **find** and **find_if**.

**Listing 23.11 FindDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
10    vector<int> intVector(values, values + 8);
11
12    ostream_iterator<int> output(cout, " ");
13    cout << "values: ";
14    copy(values, values + 8, output);
15    cout << "\nintVector: ";
```

```
 16    copy(intVector.begin(), intVector.end(), output);
 17
 18    int key;
 19    cout << "\nEnter a key: ";
 20    cin >> key;
 21    cout << "Find " << key << " in values: ";
 22    int *p = find(values, values + 8, key);
 23    if (p != values + 8)
 24      cout << "found at position " << (p - values);
 25    else
 26      cout << "not found";
 27
 28    cout << "\nFind " << key << " in intVector: ";
 29    vector<int>::iterator itr = find(intVector.begin(),
intVector.end(), key);
 30    if (itr != intVector.end())
 31      cout << "found at position " << (itr - intVector.begin());
 32    else
 33      cout << "not found";
 34
 35    return 0;
 36  }
```

***Sample output***

```
values: 1 7 3 4 3 6 1 2
intVector: 1 7 3 4 3 6 1 2
Enter a key: 4
Find 4 in values: found at position 3
Find 4 in intVector: found at position 3

values: 1 7 3 4 3 6 1 2
intVector: 1 7 3 4 3 6 1 2
Enter a key: 5
Find 5 in values: not found
Find 5 in intVector: not found
```

The **`find`** function (line 22) returns the pointer of the first element in the array that matches the **`key`**. If not found, **`p`** is **`values + 8`** (line 23). If found, **`(p – values)`** is the index of the matching element in the array.

The **`find`** function (line 29) returns the pointer of the first element in the vector that matches the **`key`**. If not found, **`itr`** is **`intVector.end()`** (line 31). If found, **`(itr – intVector.end())`** is the index of the matching element in the vector.

The **`find_end`** function is used to search a subsequence. It has two versions:

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator find_end(ForwardIterator1 beg1, ForwardIterator1 end1,
  ForwardIterator2 beg2, ForwardIterator2 end2)
```

```
template<typename ForwardIterator1, typename ForwardIterator2,
  typename boolFunction>
ForwardIterator find_end(ForwardIterator1 beg1, ForwardIterator1 end1,
  ForwardIterator2 beg2, ForwardIterator2 end2, boolFunction f)
```

Both functions search in sequence **beg1 .. end1 – 1** for a match of the entire sequence **beg2 .. end2 – 1**. If successful, return the position where the last match occurs; otherwise, return **end1**. In the first version, the elements are compared for equality; in the second version, the comparison **boolFunction(elementInFirstSequence, elementInSecondSequence)** must be **true**.

Listing 23.12 demonstrates how to use the two versions of the **find_end** function.

**Listing 23.12 FindEndDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 7, 3, 4, 3, 6, 1, 2};
10    int array2[] = {3, 6, 1};
11    vector<int> intVector(array1, array1 + 8);
12
13    ostream_iterator<int> output(cout, " ");
14    cout << "array1: ";
15    copy(array1, array1 + 8, output);
16    cout << "\nintVector: ";
17    copy(intVector.begin(), intVector.end(), output);
18
19    int *p = find_end(array1, array1 + 8, array2, array2 + 1);
20    if (p != array1 + 8)
21      cout << "\nfind {3} in array1 at position " << (p - array1);
22    else
23      cout << "\nnot found";
24
25    vector<int>::iterator itr =
26      find_end(intVector.begin(), intVector.end(), array2 + 1, array2
+ 2);
27    if (itr != intVector.end())
28      cout << "\nfind {6, 1} in intVector at position " <<
29        (itr - intVector.begin());
30    else
31      cout << "\nnot found";
32
33    return 0;
34  }
```

***Sample output***

```
array1: 1 7 3 4 3 6 1 2
intVector: 1 7 3 4 3 6 1 2
find {3} in array1 at position 4
find {6, 1} in intVector at position 5
```

The program creates two arrays and a vector (lines 9–11). The contents of these three containers are:

```
array1: {1, 7, 3, 4, 3, 6, 1, 2}
array2: {3, 6, 1}
intVector: {1, 7, 3, 4, 3, 6, 1, 2}
```

Invoking **`find_end(array1, array1 + 8, array2, array2 + 1)`** searches **`array1`** to match {**`3`**, **`6`**}. The position of the last successful match is **`4`**.

Invoking **`find_end(intVector.begin(), intVector.end(), array2 + 1, array2 + 2)`** searches **`intVector`** to match {**`6`**, **`1`**}. The position of the last successful match is **`5`**.

The function **`find_first_of`** searches the first common element in two sequences. It has two versions:

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator find_first_of(ForwardIterator1 beg1,
  ForwardIterator1 end1, ForwardIterator2 beg2,
  ForwardIterator2 end2)

template<typename ForwardIterator1, typename ForwardIterator2,
  typename boolFunction>
ForwardIterator find_first_of(ForwardIterator1 beg1,
  ForwardIterator1 end1, ForwardIterator2 beg2,
  ForwardIterator2 end2, boolFunction)
```

Both functions return a position in the first sequence if there is a match; otherwise, return **`end1`**. In the first version, the elements are compared for equality; in the second version, the comparison **`boolFunction(elementInFirstSequence, elementInSecondSequence)`** must be **`true`**.

Listing 23.13 demonstrates how to use the two versions of the **`find_first_of`** function.

**Listing 23.13 FindFirstOfDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <iterator>
5  using namespace std;
6
```

```
 7  bool greaterThan(int e1, int e2)
 8  {
 9    return e1 > e2;
10  }
11
12  int main()
13  {
14    int array1[] = {1, 7, 3, 4, 3, 6, 1, 2};
15    int array2[] = {9, 96, 21, 3, 2, 3, 1};
16    vector<int> intVector(array1, array1 + 8);
17
18    ostream_iterator<int> output(cout, " ");
19    cout << "array1: ";
20    copy(array1, array1 + 8, output);
21    cout << "\nintVector: ";
22    copy(intVector.begin(), intVector.end(), output);
23
24    int *p = find_first_of(array1, array1 + 8, array2 + 2, array2 +
4);
25    if (p != array1 + 8)
26      cout << "\nfind first of {21, 3} in array1 at position "
27        << (p - array1);
28    else
29      cout << "\nnot found";
30
31    vector<int>::iterator itr =
32      find_first_of(intVector.begin(), intVector.end(),
33        array2 + 2, array2 + 4, greaterThan);
34    if (itr != intVector.end())
35      cout << "\nfind {21, 3} in intVector at position " <<
36        (itr - intVector.begin());
37    else
38      cout << "\nnot found";
39
40    return 0;
41  }
```

***Sample output***

```
array1: 1 7 3 4 3 6 1 2
intVector: 1 7 3 4 3 6 1 2
find first of {21, 3} in array1 at position 2
find {21, 3} in intVector at position 1
```

The program creates two arrays and a vector (lines 14–16). The contents of these three containers are:

array1: {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}

array2: {**9**, **96**, **21**, **3**, **2**, **3**, **1**}

intVector: {**1**, **7**, **3**, **4**, **3**, **6**, **1**, **2**}

Invoking **find_first_of(array1, array1 + 8, array2 + 2, array2 + 4)** searches **array1** to find the first match in {**21**, **3**}, which is **3**. The position of **3** is **2** in **array1**.

Invoking **find_first_of(intVector.begin(), intVector.end(), array2 + 2, array2 + 4, greaterThan)** searches **intVector** to find the first element greater than the element in {**21**, **3**}. Element **7** in **intVector** satisfies the condition. The position of **7** in **intVector** is **1**.

***Check point***

23.9 What are the **find**, **find_if**, **find_end**, and **find_first_of** algorithms for? Do these functions return a Boolean value?

### 23.10 search and search_n

Key Point: *STL provides the **search** and **search_n** functions for searching an element in a container.*

The function **search** is similar to the function **find_end**. Both search for a subsequence. The **find_end** finds the last match, but **search** finds the first match. The **search** function has two versions:

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator search(ForwardIterator1 beg1, ForwardIterator1 end1,
  ForwardIterator2 beg2, ForwardIterator2 end2)

template<typename ForwardIterator1, typename ForwardIterator2,
  typename boolFunction>
ForwardIterator search(ForwardIterator1 beg1, ForwardIterator1 end1,
  ForwardIterator2 beg2, ForwardIterator2 end2, boolFunction)
```

Both functions return a position in the first sequence if there is a match; otherwise, they return **end1**. In the first version, the elements are compared for equality; in the second, the comparison **boolFunction(elementInFirstSequence, elementInSecondSequence)** must be **true**.

The **search_n** function searches for consecutive occurrences of a value in the sequence. The **search_n** function has two versions:

```
template<typename ForwardIterator, typename size, typename T>
ForwardIterator search_n(ForwardIterator beg,
  ForwardIterator end, size count, const T& value)

template<typename ForwardIterator, typename size,
  typename boolFunction>
ForwardIterator search_n(ForwardIterator1 beg,
  ForwardIterator1 end, size count, boolFunction f)
```

Both functions return a position of the matching element in the sequence if there is a match; otherwise, they return **end**. In the first version, the elements are compared for equality; in the second, the comparison **boolFunction(element)** must be **true**.

Listing 23.14 demonstrates how to use the functions **search** and **search_n**.

**Listing 23.14 SearchDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};
10    int array2[] = {9, 96, 4, 3, 2, 3, 1};
11    vector<int> intVector(array1, array1 + 8);
12
13    ostream_iterator<int> output(cout, " ");
14    cout << "array1: ";
15    copy(array1, array1 + 8, output);
16    cout << "\nintVector: ";
17    copy(intVector.begin(), intVector.end(), output);
18
19    int *p = search(array1, array1 + 8, array2 + 2, array2 + 4);
20    if (p != array1 + 8)
21      cout << "\nSearch {4, 3} in array1 at position "
22        << (p - array1);
23    else
24      cout << "\nnot found";
25
26    vector<int>::iterator itr =
27      search_n(intVector.begin(), intVector.end(), 2, 3);
28    if (itr != intVector.end())
29      cout << "\nSearch two occurrence of 3 in intVector at position
"
30        << (itr - intVector.begin());
31    else
32      cout << "\nnot found";
33
34    return 0;
```

```
35  }
```

***Sample output***

```
array1: 1 7 3 4 3 3 1 2
intVector: 1 7 3 4 3 3 1 2
Search {4, 3} in array1 at position 3
Search two occurrences of 3 in intVector at position 4
```

The program creates two arrays and a vector (lines 9–11). The contents of these three containers are:

```
array1: {1, 7, 3, 4, 3, 3, 1, 2}
array2: {9, 96, 4, 3, 2, 3, 1}
intVector: {1, 7, 3, 4, 3, 3, 1, 2}
```

Invoking **`search(array1, array1 + 8, array2 + 2, array2 + 4)`** searches **`array1`** to find the sequence {**`4`**, **`3`**}. The matching position is **`3`** in **`array1`**.

Invoking **`search_n(intVector.begin(), intVector.end(), 2, 3)`** searches for two consecutive **`3`**. The matching position is **`4`** in **`array1`**.

***Check point***

23.10 What are the **`search`** and **`search_n`** algorithms for? What are the differences between **`search`** and **`find_end`**?

### 23.11 `sort` and `binary_search`

Key Point: *STL provides the* ***`sort`*** *function to sort the elements in a container and the* ***`binary_search`*** *function for searching an element using binary search.*

The **`sort`** function requires random-access iterators. You can apply it to sort an array, vector, or deque, using one of the two versions:

```
template<typename randomAccessIterator>
void sort(randomAccessIterator beg, randomAccessIterator end)

template<typename randomAccessIterator, typename relationalOperator>
void sort(randomAccessIterator beg, randomAccessIterator end,
  relationalOperator op)
```

The **`binary_search`** function searches a value in a sorted sequence, using one of the two versions:

```
template<typename ForwardIterator, typename T>
bool binary_search(ForwardIterator beg,
  ForwardIterator end, const T& value)
```

```
template<typename ForwardIterator, typename T,
  typename strictWeakOrdering>
bool binary_search(ForwardIterator beg,
  ForwardIterator end, const T& value, strictWeakOrdering op)
```

NOTE:

Some STL algorithms allow you to pass a function operator. It is actually a function object, and a pointer of the object is passed to invoke an STL function. There are three kinds of function objects: relational, logic, and arithmetic, as shown in Table 23.5. The **sort** and **binary_search** algorithm require the relational operator. To use function objects, include the **<functional>** header.

Table 23.5

Function Objects

| STL function object | type | STL function object | type |
|---|---|---|---|
| equal_to<T> | Relational | plus<T> | Arithmetic |
| not_equal_to<T> | Relational | minus<T> | Arithmetic |
| greater<T> | Relational | multiplies<T> | Arithmetic |
| greater_equal<T> | Relational | divides<T> | Arithmetic |
| less<T> | Relational | modulus<T> | Arithmetic |
| less_equal<T> | Relational | negate<T> | Arithmetic |
| logical_and<T> | Logical | | |
| logical_not<T> | Logical | | |
| logical_or<T> | Logical | | |

NOTE:

Strict weak ordering operators are **less<T>** and **greater(T)**. The first version of the **binary_search** algorithm uses the **less<T>** operator for comparison, and the second specifies **less<T>** or **greater(T)**.

Listing 23.15 demonstrates how to use the functions **sort** and **binary_search**.

**Listing 23.15 SortDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
```

```
 3  #include <iterator>
 4  #include <functional>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};
10
11    ostream_iterator<int> output(cout, " ");
12    cout << "Before sort, array1: ";
13    copy(array1, array1 + 8, output);
14
15    sort(array1, array1 + 8);
16
17    cout << "\nAfter sort, array1: ";
18    copy(array1, array1 + 8, output);
19
20    cout << (binary_search(array1, array1 + 8, 4) ?
21      "\n4 is in array1" : "\n4 is not in array1");
22
23    sort(array1, array1 + 8, greater_equal<int>());
24
25    cout << "\nAfter sort with function operator(>=), array1: ";
26    copy(array1, array1 + 8, output);
27
28    cout << (binary_search(array1, array1 + 8, 4,
29      greater_equal<int>()) ?
30      "\n4 is in array1" : "\n4 is not in array1");
31
32    return 0;
33  }
```

***Sample output***

```
Before sort, array1: 1 7 3 4 3 3 1 2
After sort, array1: 1 1 2 3 3 3 4 7
4 is in array1
After sort with function operator(>), array1: 7 4 3 3 3 2 1 1
4 is in array1
```

The default function operator for **`sort`** and **`binary_search`** is **`less_equal<T>()`**. Invoking **`sort(array1, array1 + 8, greater_equal<int>())`** (line 23) sorts the array using the **`greater_equal<int>()`** function object. Since the elements are ordered in decreasing order, you invoke **`binary_search(array1, array1 + 8, 4, greater_equal<int>())`** (lines 28–29) to perform binary search to find element **`4`** in **`array1`**.

***Check point***

23.11 What are the return types for these two functions? What iterator types are needed for **`sort`** and **`binary search`**? Can you apply the **`sort`** algorithm on a list?

### 23.12 adjacent_find, merge, and inplace_merge

Key Point: *STL provides the* ***adjacent_find*** *function to find adjacent elements of the same value and the* ***merge*** *and* ***inplace_merge*** *functions to merge two sorted sequences.*

The **adjacent_find** function looks for first occurrence of adjacent elements of equal value or satisfying **boolFunction(element)**, using the following syntax:

```
template<typename ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator beg,
  ForwardIterator end)

template<typename ForwardIterator, typename boolFunction>
ForwardIterator adjacent_find(ForwardIterator beg,
  ForwardIterator end, boolFunction f)
```

The **adjacent_find** function returns the iterator that points to the first element in the matching sequence. If not found, it returns **end**.

The **merge** function merges two sorted sequences into a new sequence, using the following syntax:

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator merge(InputIterator1 beg1,
  InputIterator1 end1, InputIterator2 beg2,
  InputIterator2 end2, OutputIterator targetPosition)

template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator, typename relationalOperator>
OutputIterator merge(InputIterator1 beg1,
  InputIterator1 end1, InputIterator2 beg2,
  InputIterator2 end2, OutputIterator targetPosition,
  relationalOperator)
```

The **inplace_merge** function merges the first part of the sequence with the second part; assume that the two parts contain sorted consecutive elements. The syntax is:

```
template<typename BidirectionalIterator>
void inplace_merge(bidirectionalIterator beg,
  bidirectionalIterator middle, bidirectionalIterator end)

template<typename BidirectionalIterator, typename relationalOperator>
void inplace_merge(bidirectionalIterator beg,
  bidirectionalIterator middle, bidirectionalIterator end,
  relationalOperator)
```

The function merges the sorted consecutive sequences **beg..middle-1** with **middle..end-1**, and the sorted sequence is stored in the original sequence. Thus, this is called *inplace merge*.

Listing 23.16 demonstrates how to use the functions **adjacent_find**, **merge**, and **inplace_merge**.

**Listing 23.16 MergeDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};
10    list<int> intList(8);
11
12    ostream_iterator<int> output(cout, " ");
13    cout << "array1: ";
14    copy(array1, array1 + 8, output);
15
16    sort(array1, array1 + 3);
17    sort(array1 + 3, array1 + 8);
18    cout << "\nafter sort partial arrays, array1: ";
19    copy(array1, array1 + 8, output);
20
21    merge(array1, array1 + 3, array1 + 3, array1 + 8,
intList.begin());
22    cout << "\nafter inplace merger, array1: ";
23    copy(intList.begin(), intList.end(), output);
24
25    inplace_merge(array1, array1 + 3, array1 + 8);
26    cout << "\nafter inplace merger, intList: ";
27    copy(array1, array1 + 8, output);
28
29    return 0;
30  }
```

***Sample output***

```
array1: 1 7 3 4 3 3 1 2
after sort partial arrays, array1: 1 3 7 1 2 3 3 4
after merger, intList: 1 1 2 3 3 3 4 7
after inplace merger, array1: 1 1 2 3 3 3 4 7
```

The program creates an array and a list (lines 9–10). The contents of the array are:

```
array1: {1, 7, 3, 4, 3, 3, 1, 2}
```

After sorting {**1**, **7**, **3**}, and {**4**, **3**, **3**, **1**, **2**} (lines 16–17), the array becomes:

```
array1: {1, 3, 7, 1, 2, 3, 3, 4}
```

After merging {**1**, **3**, **7**} and {**1**, **2**, **3**, **3**, **4**} into **intList** (line 21), **intList** becomes

```
intList: {1, 1, 2, 3, 3, 3, 4, 7}
```

After inplace merging {**1**, **3**, **7**} and {**1**, **2**, **3**, **3**, 4} (line 25), **array1** becomes

```
intList: {1, 1, 2, 3, 3, 3, 4, 7}
```

***Check point***

23.12 What are the **adjacent_find**, **merge**, and **inplace_merge** algorithms for?

Show the output of the following code:

```
int values[] = {1, 2, 3, 4, 4, 5, 1, 1};
int* p = adjacent_find(values, values + 8);

ostream_iterator<int> output(cout, " ");
cout << "values: ";
copy(p, values + 8, output);
```

### 23.13 reverse and reverse_copy

Key Point: STL provides the **reverse** and **reverse_copy** functions to reverse the elements in a sequence.

The **reverse** function reverses the elements in a sequence. The **reverse_copy** function copies the elements in one sequence to the other in reverse order. The **reverse_copy** function does not change the contents in the source container. The syntax of these functions is:

```
template<typename BidirectionalIterator>
void reverse(BidirectionalIterator beg,
  BidirectionalIterator end)

template<typename BidirectionalIterator, typename OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator beg,
  BidirectionalIterator end, OutputIterator targetPosition)
```

Listing 23.17 demonstrates how to use the functions **reverse** and **reverse_copy**.

**Listing 23.17 ReverseDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};
10    list<int> intList(8);
```

```
11
12    ostream_iterator<int> output(cout, " ");
13    cout << "array1: ";
14    copy(array1, array1 + 8, output);
15
16    reverse(array1, array1 + 8);
17    cout << "\nafter reverse arrays, array1: ";
18    copy(array1, array1 + 8, output);
19
20    reverse_copy(array1, array1 + 8, intList.begin());
21    cout << "\nafter reverse_copy, array1: ";
22    copy(array1, array1 + 8, output);
23    cout << "\nafter reverse_copy, intList: ";
24    copy(intList.begin(), intList.end(), output);
25
26    return 0;
27  }
```

***Sample output***

```
array1: 1 7 3 4 3 3 1 2
after reverse arrays, array1: 2 1 3 3 4 3 7 1
after reverse_copy, array1: 2 1 3 3 4 3 7 1
after reverse_copy, intList: 1 7 3 4 3 3 1 2
```

***Check point***

23.13 What are the **reverse** and **reverse_copy** algorithms for? Does the **reverse_copy** algorithm change the contents of the original sequence?

### 23.14 rotate and rotate_copy

Key Point: STL provides the **rotate** and **rotate_copy** functions to rotate the elements in a sequence.

The **rotate** function rotates the elements in a sequence, using the syntax:

```
template<typename ForwardIterator>
void rotate(ForwardIterator beg, ForwardIterator newBeg,
  ForwardIterator end)
```

The element specified by **newBeg** becomes the first element in the sequence after the rotate.

The **rotate_copy** function is similar to **rotate** except that it copies the result to a target sequence, using the syntax:

```
template<typename ForwardIterator, typename OutputIterator>
OutputIterator rotate_copy(ForwardIterator beg, ForwardIterator newBeg,
  ForwardIterator end, OutputIterator targetPosition)
```

Listing 23.18 demonstrates how to use the functions **reverse**, and **reverse_copy**.

**Listing 23.18 RotateDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <list>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
10    list<int> intList(8);
11
12    ostream_iterator<int> output(cout, " ");
13    cout << "array1: ";
14    copy(array1, array1 + 8, output);
15
16    rotate(array1, array1 + 3, array1 + 8);
17    cout << "\nafter rotate arrays, array1: ";
18    copy(array1, array1 + 8, output);
19
20    rotate_copy(array1, array1 + 1, array1 + 8, intList.begin());
21    cout << "\nafter rotate_copy, array1: ";
22    copy(array1, array1 + 8, output);
23    cout << "\nafter rotate_copy, intList: ";
24    copy(intList.begin(), intList.end(), output);
25
26    return 0;
27  }
```

***Sample output***

```
array1: 1 2 3 4 5 6 7 8
after rotate arrays, array1: 4 5 6 7 8 1 2 3
after rotate_copy, array1: 4 5 6 7 8 1 2 3
after rotate_copy, intList: 5 6 7 8 1 2 3 4
```

The program creates an array and a list (lines 9–10). The contents of the array is:

```
array1: {1, 2, 3, 4, 5, 6, 7, 8}
```

The pointer **`array1 + 3`** points to **`4`**, so after invoking **`rotate(array1, array1 + 3, array1 + 8)`**, **`array1`** becomes

```
array1: {4, 5, 6, 7, 8, 1, 2, 3}
```

Now the pointer **`array1 + 1`** points **`5`**, so after invoking **`rotate_copy(array1, array1 + 1, array1 + 8, intList.begin())`**, **`intList`** becomes

```
intList: {5, 6, 7, 8, 1, 2, 3, 4}
```

***Check point***

23.14 What are the **`rotate`** and **`rotate_copy`** algorithms for? Show the output of the following code:

```
int values[] = {1, 2, 3, 4, 4, 5, 1, 1};
```

```
rotate(values, values + 5, values + 8);

ostream_iterator<int> output(cout, " ");
cout << "values: ";
copy(values, values + 8, output);
```

### 23.15 `swap`, `iter_swap`, and `swap_ranges`

Key Point: *STL provides the* ***`swap`****,* ***`iter_swap`****, and* ***`swap_ranges`*** *functions to swap the elements in a sequence.*

The syntax of these functions are defined as follows:

```
template<typename T>
void swap(T& value1, T& value2)

template<typename ForwardIterator1, typename ForwardIterator2>
void iter_swap(ForwardIterator p1, ForwardIterator p2)

template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator swap_ranges(ForwardIterator1 beg1,
  ForwardIterator1 end1, ForwardIterator2 beg2)
```

The **`swap`** function swaps the values in two variables. The **`iter_swap`** function swaps the values pointed to by the iterators. The **`swap_ranges`** function swaps two sequences.

Listing 23.19 demonstrates how to use these three functions.

**Listing 23.19 SwapDemo.cpp**

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
  int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
  ostream_iterator<int> output(cout, " ");
  cout << "array1: ";
  copy(array1, array1 + 8, output);

  cout << "\nafter swap variables, array1: ";
  swap(array1[0], array1[1]);
  copy(array1, array1 + 8, output);

  cout << "\nafter swap via pointers, array1: ";
  iter_swap(array1 + 2, array1 + 3);
  copy(array1, array1 + 8, output);

```

```
21    cout << "\nafter swap ranges, array1: ";
22    swap_ranges(array1, array1 + 4, array1 + 4);
23    copy(array1, array1 + 8, output);
24
25    return 0;
26  }
```

***Sample output***

```
array1: 1 2 3 4 5 6 7 8
after swap variables, array1: 2 1 3 4 5 6 7 8
after swap via pointers, array1: 2 1 4 3 5 6 7 8
after swap ranges, array1: 5 6 7 8 2 1 4 3
```

Invoking **`swap(array1[0], array1[1])`** swaps **`array1[0]`** with **`array1[1]`** (line 14).

Invoking **`iter_swap(array1 + 2, array1 + 3)`** swaps the elements pointed by **`array1 + 2`** and **`array1 + 3`** (line 18).

Invoking **`swap_ranges(array1, array1 + 4, array1 + 4)`** swaps the elements in **`array1..array1 + 3`** with the elements in **`array1 + 4..array1 + 7`** (line 22).

***Check point***

23.15 What are the **`swap`**, **`iter_swap`**, and **`swap_ranges`** algorithms for?

### 23.16 `count` and `count_if`

Key Point: *STL provides the* ***`count`*** *and* ***`count_if`*** *functions to count the occurrences of the elements in a sequence.*

The **`count`** function counts the occurrence of a given value in the sequence, using the following syntax:

```
template<typename InputIterator, typename T>
int count(InputIterator beg, InputIterator end, const T& value)
```

The **`count_if`** function counts the occurrence of the elements such that **`boolFunction(element)`** is true, using the following syntax:

```
template<typename InputIterator, typename boolFunction>
int count_if(InputIterator beg, InputIterator end, boolFunction f)
```

Listing 23.20 demonstrates how to use these functions.

**Listing 23.20 CountDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  using namespace std;
 4
 5  bool greaterThan1(int value)
 6  {
 7    return value > 1;
 8  }
 9
10  int main()
11  {
12    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
13
14    cout << "The number of 1's in array1: " <<
15      count(array1, array1 + 8, 1) << endl;
16
17    cout << "The number of elements > 1 in array1: " <<
18      count_if(array1, array1 + 8, greaterThan1) << endl;
19
20    return 0;
21  }
```

***Sample output***

```
The number of 1's in array1: 1
The number of elements > 1 in array1: 7
```

***Check point***

23.16 What are the **`count`** and **`count_if`** algorithms for?

### 23.17 `max_element` and `min_element`

Key Point: *STL provides the* ***`max_element`*** *and* ***`min_element`*** *functions to return the maximum and minimum elements in a sequence.*

You are already familiar with the **`max`** and **`min`** functions. You can use the **`max_element`** and **`min_element`** to obtain the maximum element and minimum element in a sequence. The functions are defined as follows:

```
template<typename ForwardIterator>
ForwardIterator max_element(ForwardIterator beg,
  ForwardIterator end)

template<typename ForwardIterator>
ForwardIterator min_element(ForwardIterator beg,
  ForwardIterator end)
```

Listing 23.21 gives an example of how to use these functions.

**Listing 23.21 MaxMinDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
```

```
 3  using namespace std;
 4
 5  int main()
 6  {
 7    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
 8
 9    cout << "The max element in array1: " <<
10      *max_element(array1, array1 + 8) << endl;
11
12    cout << "The min element in array1: " <<
13      *min_element(array1, array1 + 8) << endl;
14
15    return 0;
16  }
```

***Sample output***

```
The max element in array1: 8
The min element in array1: 1
```

***Check point***

23.17 What are the **`max`** and **`max_element`** algorithms for?

### 23.18 `random_shuffle`

Key Point: *The* ***`random_shuffle`*** *function randomly reorders the elements in a sequence.*

The **`random_shuffle`** function has two versions:

```
template<typename randomAccessIterator>
void random_shuffle(randomAccessIterator beg,
  randomAccessIterator end)


template<typename randomAccessIterator>
void random_shuffle(randomAccessIterator beg,
  randomAccessIterator end, RandomNumberGenerator& rand)
```

Listing 23.22 gives an example of how to use this function.

**Listing 23.22 ShuffleDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <iterator>
 4  #include <ctime>
 5  using namespace std;
 6
 7  int randGenerator(int aRange)
 8  {
 9    srand(time(0));
10    return rand() % aRange;
11  }
12
13  int main()
14  {
```

```
15    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
16    random_shuffle(array1, array1 + 8);
17    cout << "After random shuffle, array1: ";
18    ostream_iterator<int> output(cout, " ");
19    copy(array1, array1 + 8, output);
20
21    int array2[] = {1, 2, 3, 4, 5, 6, 7, 8};
22    random_shuffle(array2, array2 + 8, randGenerator);
23    cout << "\nAfter random shuffle, array2: ";
24    copy(array2, array2 + 8, output);
25
26    return 0;
27  }
```

***Sample output***

```
After random shuffle, array1: 2 6 4 3 7 5 1 8
After random shuffle, array2: 6 5 4 3 2 8 7 1
```

The program creates **array1** (line 15) and invokes the first version of the **random_shuffle** function to randomly shuffle the elements in the array. The first version of the **random_shuffle** function uses the internal default random number generator, which generates random numbers in a fixed sequence. Everytime, you run the code, you will get the same sequence of the elements in **array1** after shuffling.

The program creates **array2** (line 21) and invokes the second version of the **random_shuffle** function to randomly shuffle the elements in the array. The second version of the **random_shuffle** function uses the custom random number generator, which generates random numbers in a random sequence because the random number seed is set randomly (line 9). Everytime, you run the code, you will get a different sequence of the elements in **array2** after shuffling.

***Check point***

23.18 What is the **random_shuffle** algorithm for?

### 23.19 for_each and transform

Key Point: *STL provides the* ***for_each*** *and* ***transform*** *functions apply a function on each element in a sequence.*

The **for_each** function is used to process each element in a sequence by applying a function, using the following syntax:

```
template<typename InputIterator, typename function>
void for_each(InputIterator beg, InputIterator end, function f)
```

You can use the **transform** function to apply a function on each element in the sequence and copy the result to a target sequence. The function is defined as follows:

```
template<typename InputIterator, typename OutputIterator,
  typename function>
OutputIteration transform(InputIterator beg,
  InputIterator end, OutputIterator targetPosition, function f)
```

Listing 23.23 demonstrates how to use these functions.

**Listing 23.23 ForEachDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <iterator>
 5  using namespace std;
 6
 7  void display(int &value)
 8  {
 9    cout << value << " ";
10  }
11
12  int square(int &value)
13  {
14    return value * value;
15  }
16
17  int main()
18  {
19    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
20    cout << "array1: ";
21    for_each(array1, array1 + 8, display);
22
23    vector<int> intVector(8);
24    transform(array1, array1 + 8, intVector.begin(), square);
25    cout << "\nintVector: ";
26    for_each(intVector.begin(), intVector.end(), display);
27
28    return 0;
29  }
```

***Sample output***

```
array1: 1 2 3 4 5 6 7 8
intVector: 1 4 9 16 25 36 49 64
```

The **display** function (lines 7–10) displays a number to the console. Invoking **for_each(array1, array1 + 8, display)** (line 21) applies the **display** function to each element in the sequence. Thus, all the elements in **array1** are displayed.

The **square** function (lines 12–15) returns the square of a number. Invoking **transform(array1, array1 + 8, intVector.begin(), square)** (line 24) applies the **square** function to each element in the sequence and copies the new sequence to **intVector**.

***Check point***

23.19 What are the **for_each** and **transform** algorithms for?

## 23.20 includes, set_union, set_difference, set_intersection, and set_symmetric_difference

Key Point: *STL provides the* ***includes***, ***set_union***, ***set_difference***, ***set_intersection***, *and* ***set_symmetric_difference*** *functions for set operations.*

The STL supports the set operations for testing subset, union, difference, intersect, and symmetric difference. All these functions require that the elements in the sequences already are sorted.

The **includes** function returns **true** if the elements in the first sequence contain the elements in the second sequence.

```
template<typename InputIterator1, typename InputIterator2>
bool includes(InputIterator1 beg1, InputIterator1 end1,
  InputIterator2 beg2, InputIterator2 end2)
```

The **set_union** function obtains the elements that belong to either sequence.

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator set_union(InputIterator1 beg1, InputIterator1 end1,
  InputIterator2 beg2, InputIterator2 end2,
  OutputIterator targetPosition)
```

The **set_difference** function obtains the elements that belong to the first sequence, but not to the second.

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator set_difference(InputIterator beg1,
  InputIterator end1, InputIterator beg2, InputIterator end2,
  OutputIterator targetPosition)
```

The **set_intersection** function obtains the elements that appear in both sequences.

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator set_intersection(InputIterator beg1,
  InputIterator end1, InputIterator beg2, InputIterator end2,
  OutputIterator targetPosition)
```

The **set_symmetric_difference** function obtains the elements that appear in either sequence, but not in both.

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator set_symmetric_difference(InputIterator beg1,
  InputIterator end1, InputIterator beg2, InputIterator end2,
  OutputIterator targetPosition)
```

Suppose **array1** and **array2** are given as follows:

| array1 | array2 |
|---|---|
| {1, 2, 3, 4, 5, 6, 7, 8} | {1, 3, 6, 9, 12} |

Their set operations are shown below:

| Operation | result |
|---|---|
| array1 union array2 | {1, 2, 3, 4, 5, 6, 7, 8, 9, 12} |
| array1 difference array2 | {2, 4, 5, 7, 8} |
| array1 intersection array2 | {1, 3, 6} |
| array1 symmetric_diff array2 | {2, 4, 5, 7, 8, 9, 12} |

NOTE:

The set functions return an iterator that points to the position after the last element in the target.

Listing 23.24 demonstrates how to use set functions.

**Listing 23.24 SetOperationDemo.cpp**

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <vector>
 4  #include <iterator>
 5  using namespace std;
 6
 7  int main()
 8  {
 9    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
10    int array2[] = {1, 3, 6, 9, 12};
11    vector<int> intVector(15);
12
13    ostream_iterator<int> output(cout, " ");
14    cout << "array1: ";
15    copy(array1, array1 + 8, output);
16    cout << "\narray2: ";
17    copy(array2, array2 + 5, output);
18
```

```
19    bool isContained =
20      includes(array1, array1 + 8, array2, array2 + 3);
21    cout << (isContained ? "\n{1, 3, 6} is a subset of array1" :
22      "\n{1, 3, 6} is not a subset of array1");
23
24    vector<int>::iterator last = set_union(array1, array1 + 8,
25      array2, array2 + 5, intVector.begin());
26    cout << "\nAfter union, intVector: ";
27    copy(intVector.begin(), last, output);
28
29    last = set_difference(array1, array1 + 8,
30      array2, array2 + 5, intVector.begin());
31    cout << "\nAfter difference, intVector: ";
32    copy(intVector.begin(), last, output);
33
34    last = set_intersection(array1, array1 + 8,
35      array2, array2 + 5, intVector.begin());
36    cout << "\nAfter intersection, intVector: ";
37    copy(intVector.begin(), last, output);
38
39    last = set_symmetric_difference(array1, array1 + 8,
40      array2, array2 + 5, intVector.begin());
41    cout << "\nAfter symmetric difference, intVector: ";
42    copy(intVector.begin(), last, output);
43
44    return 0;
45  }
```

***Sample output***

```
array1: 1 2 3 4 5 6 7 8
array2: 1 3 6 9 12
{1, 3, 6} is a subset of array1
After union, intVector: 1 2 3 4 5 6 7 8 9 12
After difference, intVector: 2 4 5 7 8
After intersection, intVector: 1 3 6
After symmetric difference, intVector: 2 4 5 7 8 9 12
```

The program creates two arrays and a vector (lines 9–11).

```
array1: {1, 2, 3, 4, 5, 6, 7, 8}
array2: {1, 3, 6, 9, 12}
```

Invoking **`includes(array1, array1 + 8, array2, array2 + 3)`** (line 20) returns **`true`**, because {**`1`**, **`3`**, **`6`**} is a subset of **`array1`**.

Invoking **`set_union(array1, array1 + 8, array2, array2 + 5, intVector.begin())`** (lines 24–25) obtains the union of **`array1`** and **`array2`** in **`intVector`**. **`intVector`** becomes

```
intVector: {1, 2, 3, 4, 5, 6, 7, 8, 9, 12}
```

Invoking **`set_difference(array1, array1 + 8, array2, array2 + 5, intVector.begin())`** (lines 29–30) obtains the difference between **`array1`** and **`array2`** in **`intVector`**. **`intVector`** becomes

```
intVector: {2, 4, 5, 7, 8}
```

Invoking **`set_intersection(array1, array1 + 8, array2, array2 + 5, intVector.begin())`** (lines 34–35) obtains the intersection between **`array1`** and **`array2`** in **`intVector`**. **`intVector`** becomes

```
intVector: {1, 3, 6}
```

Invoking **`set_symmetric_difference(array1, array1 + 8,  array2, array2 + 5, intVector.begin())`** (lines 39–40) obtains the symmetric difference between **`array1`** and **`array2`** in **`intVector`**. **`intVector`** becomes

```
intVector: {2, 4, 5, 7, 8, 9, 12}
```

NOTE:

The set operations store the contents to a result container. It must be large enough to hold the result. So, **`intVector`** is declared with 15 elements (line 11). Moreover, the number of elements in **`intVector`** does not change in this program. The result elements are between **`intVector.begin()`** and **`intVector.end()`**.

***Check point***

23.20 Suppose **`array1`** is {**1**, **2**, **3**, **4**, **5**} and **`array2`** is {**2**, **4**, **8**, **9**, **10**}. Show the union, difference, intersection, and symmetric difference of these two arrays.

### 23.21 `accumulate`, `adjacent_difference`, `inner_product`, and `partial_sum`

Key Point: *The STL supports the mathematical functions* ***`accumulate`***, ***`adjacent_difference`***, ***`inner_product`***, *and* ***`partial_sum`***. *They are defined in the* ***`<numeric>`*** *header.*

The **`accumulate`** function has two versions.

```
template<typename InputIterator, typename T>
```

```
T accumulate(InputIterator beg, InputIterator end, T initValue)

template<typename InputIterator, typename T,
  typename arithmeticOperator>
T accumulate(InputIterator beg, InputIterator end, T initValue,
  arithmeticOperator op)
```

The first version returns the sum of all the elements and the **initValue**. The second version applies the arithmetic operators (e.g., multiplication) on the **initValue** with all the elements and returns the result. For example,

| array1 | result of accumulate(array1, array1 + 5, 0) |
|---|---|
| {1, 2, 3, 4, 5} | 15 |

| array1 | result of accumulate(array1, array1 + 5, 1, multiplies<int>()) |
|---|---|
| {1, 2, 3, 4, 5} | 120 |

The **adjacent_difference** function has two versions.

```
template<typename InputIterator, typename T>
OutputIterator adjacent_difference(InputIterator beg,
  InputIterator end, OutputIterator targetPosition)

template<typename InputIterator, typename T,
  typename arithmeticOperator>
OutputIterator adjacent_difference(InputIterator beg,
  InputIterator end, OutputIterator targetPosition,
  arithmeticOperator op)
```

The first version creates a sequence of elements in which the first element is the same as the first element in the input sequence, and each subsequent element is the difference between the current element and the previous one. The second version is the same as the first, except that the specified arithmetic operator is applied to replace the subtraction operator. For example,

| array1 | result of adjacent_difference(array1, array1 + 5, intVector.begin()) |
|---|---|
| {1, 2, 3, 4, 5} | {1, 1, 1, 1, 1} |

The **inner_product** function has two versions.

```
template<typename InputIterator1, typename InputIterator2,
  typename T>
T inner_product(InputIterator1 beg1,
  InputIterator1 end1, InputIterator2 beg2, T initValue)
```

```
template<typename InputIterator1, typename InputIterator2,
  typename T, typename arithmeticOperator1,
  typename arithmeticOperator2>
T inner_product(InputIterator1 beg1,
  InputIterator1 end1, InputIterator2 beg2, T initValue,
  arithmeticOperator1 op1, arithmeticOperator2 op2)
```

The inner product of two sequences **{a1, a2, ..., ai}** and **{b1, b2, ..., bi}** is defined as

**a1 * b1 + a2 * b2 + ... + ai * bi**

The first version returns the sum of **initValue** and the inner product of the sequences. The second version is the same as the first, except that the default addition operator is replaced by **op1** and the multiplication operator is replaced by **op2**. For example,

| array1 | result of inner_product(array1, array1 + 5, array1, 0) |
|---|---|
| {1, 2, 3, 4, 5} | 55 |

The **partial_sum** function has two versions.

```
template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator>
OutputIterator partial_sum(InputIterator1 beg1,
  InputIterator1 end1, OutputIterator2 beg2)

template<typename InputIterator1, typename InputIterator2,
  typename OutputIterator, typename arithmeticOperator>
OutputIterator partial_sum(InputIterator beg1,
  InputIterator end1, OutputIterator targetPosition
  arithmeticOperator op)
```

The first version creates a sequence in which each element is the sum of all the preceding elements. The second version is the same as the first, except that the default addition operator is replaced by **op**. For example,

| array1 | result of partial_sum(array1, array1 + 5, intVector.begin()) |
|---|---|
| {1, 2, 3, 4, 5} | {1, 3, 6, 10, 15} |

Listing 23.25 demonstrates how to use the mathematical functions.

**Listing 23.25 MathOperationDemo.cpp**

```
1  #include <iostream>
2  #include <algorithm>
3  #include <numeric>
4  #include <vector>
5  #include <iterator>
6  #include <functional>
```

```
 7  using namespace std;
 8
 9  int main()
10  {
11    int array1[] = {1, 2, 3, 4, 5};
12    vector<int> intVector(5);
13
14    ostream_iterator<int> output(cout, " ");
15    cout << "array1: ";
16    copy(array1, array1 + 5, output);
17
18    cout << "\nSum of array1: " <<
19      accumulate(array1, array1 + 5, 0) << endl;
20
21    cout << "Product of array1: " <<
22      accumulate(array1, array1 + 5, 1, multiplies<int>()) << endl;
23
24    vector<int>::iterator last =
25      adjacent_difference(array1, array1 + 5, intVector.begin());
26    cout << "After adjacent difference, intVector: ";
27    copy(intVector.begin(), last, output);
28
29    cout << "\nInner product of array1 * array1 is " <<
30     inner_product(array1, array1 + 5, array1, 0);
31
32    last = partial_sum(array1, array1 + 5, intVector.begin());
33    cout << "\nAfter partial sum, intVector: ";
34    copy(intVector.begin(), last, output);
35
36    return 0;
37  }
```

***Sample output***

```
array1: 1 2 3 4 5
Sum of array1: 15
Product of array1: 120
After adjacent difference, intVector: 1 1 1 1 1
Inner product of array1 * array1 is 55
After partial sum, intVector: 1 3 6 10 15
```

The program creates an array and a vector (lines 10–11).

```
array1: {1, 2, 3, 4, 5}
```

Invoking **`accumulate(array1, array1 + 5, 0)`** (line 19) returns the sum of all the elements in **`array1`**.

Invoking **`accumulate(array1, array1 + 5, 1, multiplies<int>())`** (line 22) returns the multiplication of all the elements in **`array1`**.

Invoking **`adjacent_difference(array1, array1 + 5, intVector.begin())`** (line 25) obtains a sequence for the adjacent difference of **`array1`** in **`intVector`**.

Invoking `inner_product(array1, array1 + 5, array1, 0)` (line 30) obtains the inner product of `array1` and `array1`.

Invoking `partial_sum(array1, array1 + 5, intVector.begin())` (line 32) obtains a sequence for the partial sum of `array1` in `intVector`.

***Check point***

23.21 Suppose `array1` is {`1`, `2`, `3`, `4`, `5`} and `array2` is {`2`, `4`, `8`, `9`, `10`}. Show the `accumulate`, `adjacent_difference`, and `partial_sum` for `array1`. What is the inner product of `array1` and `array2`?

**Key Terms**

- `accumulate` **algorithm**
- `adjacent_find` **algorithm**
- `adjacent_difference` **algorithm**
- `binary_search` **algorithm**
- `copy` **algorithm**
- `count` **algorithm**
- `count_if` **algorithm**
- `fill` **algorithm**
- `fill_n` **algorithm**
- `find` **algorithm**
- `find_end` **algorithm**
- `find_first_of` **algorithm**
- `find_if` **algorithm**
- `for_each` **algorithm**
- **function object**
- `generate` **algorithm**

- `generate_n` **algorithm**
- **heap algorithms**
- `inplace_merge` **algorithm**
- `inner_product` **algorithm**
- `includes` **algorithm**
- `iter_swap` **algorithm**
- `max_element` **algorithm**
- `merge` **algorithm**
- `min_element` **algorithm**
- **modifying STL algorithms**
- **nonmodifying STL algorithms**
- **numeric STL algorithms**
- `partial_sum` **algorithm**
- `random_shuffle` **algorithm**
- `remove` **algorithm**
- `remove_copy` **algorithm**
- `remove_copy_if` **algorithm**
- `remove_if` **algorithm**
- `replace` **algorithm**
- `replace_if` **algorithm**
- `replace_copy` **algorithm**
- `replace_copy_if` **algorithm**
- `reverse` **algorithm**
- `reverse_copy` **algorithm**
- `rotate` **algorithm**
- `rotate_copy` **algorithm**
- `search` **algorithm**

- **`search_n` algorithm**
- **`sort` algorithm**
- **`swap` algorithm**
- **`swap_range` algorithm**
- **`set_difference` algorithm**
- **`set_intersection` algorithm**
- **`set_symmetric_difference` algorithm**
- **`set_union` algorithm**
- **`transform` algorithm**

**Chapter Summary**

1. The STL separates the algorithms from the containers. This enables the algorithms to be applied generically to all containers through iterators. The STL makes the algorithms and containers easy to maintain.
2. The STL provides approximately 80 algorithms. They can be classified into four groups: nonmodifying algorithms, modifying algorithms, numeric algorithms, and heap algorithms.
3. All the algorithms operate through iterators. Many algorithms operate on a sequence of elements pointed to by two iterators. The first iterator points to the first element of the sequence, and the second points to the element after the last element of the sequence.
4. The **`copy`** function can be used to copy elements in a sequence from one container to another.
5. The functions **`fill`** and **`fill_n`** can be used to fill a container with a specified value.
6. The functions **`generate`** and **`generate_n`** fill a container with a value returned from a function.
7. The functions **`remove`** and **`remove_if`** remove the elements from a sequence that matches some criteria. The functions **`remove_copy`** and **`remove_copy_if`** are similar to **`remove`** and **`remove_if`** except that they copy the result to a target sequence.

8. The functions **replace** and **replace_if** replace all occurrences of a given value with a new value in a sequence. The functions **replace_copy** and **replace_copy_if** are similar to **replace** and **replace_if** except that they copy the result to a target sequence.
9. The functions **find**, **find_if**, **find_end**, and **find_first_of** can be used to find the elements in sequence.
10. The functions **search** and **search_n** search for a subsequence.
11. The **sort** function requires random-access iterators. You can apply it to sort an array, vector, or deque.
12. The **adjacent_find** function looks for the first occurrence of adjacent elements of equal value or satisfying **boolFunction(element)**.
13. The **merge** function merges two sorted sequences into a new sequence.
14. The **inplace_merge** function merges the first part of the sequence with the second part; assume that the two parts contain sorted consecutive elements.
15. The **reverse** function reverses the elements in a sequence. The **reverse_copy** function copies the elements in one sequence to the other in reverse order.
16. The **rotate** function rotates the elements in a sequence. The **rotate_copy** function is similar to **rotate**, except that it copies the result to a target sequence.
17. The **swap** function swaps the values in two variables. The **iter_swap** function swaps the values pointed to by the iterators. The **swap_range** function swaps two sequences.
18. The **count** function counts the occurrence of a given value in the sequence. The **count_if** function counts the occurrence of the elements such that **boolFunction(element)** is true.
19. The functions **max_element** and **min_element** obtain the maximum element and minimum element in a sequence.
20. The **random_shuffle** function randomly reorders the elements in a sequence.
21. The **for_each** function is used to process each element in a sequence by applying a function. The **transform** function is used to apply a function on each element in the sequence and copy the result to a target sequence.

22. The STL supports the set operations **includes**, **set_union**, **set_difference**, **set_intersection**, and **set_symmetric_difference**. All these functions require that the elements in the sequences already are sorted.
23. The STL supports the mathematical functions **accumulate**, **adjacent_difference**, **inner_product**, and **partial_sum**. They are defined in the **<numeric>** header.

**Quiz**

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

**Programming Exercises**

23.1 Create an array of double values with five numbers: **1.3**, **2.4**, **4.5**, **6.7**, **9.0**. Use the **fill** function to fill the first three elements with **5.5**. Use the **fill_n** function to fill the first four elements with **6.9**.

23.2 Create a **deque** with five numbers: **1.3**, **2.4**, **4.5**, **6.7**, **9.0**. Use the **generate** function to fill random numbers in the deque. Use the **generate_n** function to fill random numbers in the deque.

23.3 Create an **array** with these numbers: **1.3**, **2.4**, **4.5**, **6.7**, **4.5**, **9.0**. Use the **remove** function to remove all the elements with value **4.5**. Use the **remove_if** function to remove all the elements that are less than **2.0**. Use the **remove_copy** function to copy all the elements except **6.7** to a list. Use the **remove_copy_if** function to copy all the elements except those that are greater than **4.0** to a list.

23.4 Create an **array** with these numbers: **2.4**, **1.3**, **2.4**, **4.5**, **6.7**, **4.5**, **9.0**. Use the **replace** function to replace all the occurrences of **2.4** with **9.9**. Use the **replace_if** function to replace all the elements that are less than **2.0** with **12.5**. Use the **replace_copy** function to replace all occurrences

of **6.7** with **9.7** and copy all the sequence to a vector. Use the **replace_copy_if** function to replace all the elements that are greater than or equal to **1.3** with **747** and copy the sequence to a vector.

23.5 Create an **array** with these numbers: **2.4**, **1.3**, **2.4**, **4.5**, **6.7**, **4.5**, **9.0**. Use the **find** function to find the position of **4.5** in the array. Use the **find_if** function to find the position of the first element that is less than **2**. Use the **find_end** function to find the position of the sequence {**2.4**, **4.5**} in the array. Use the **find_first_of** function to find the position of the first common element in the array and the list {**34**, **55**, **2.4**, **4.5**}.

23.6 Create an **array** with these numbers: **2.4**, **1.3**, **2.4**, **2.4**, **4.5**, **6.7**, **4.5**, **9.0**. Use the **search** function to find the position of the sequence {**2.4**, **4.5**} in the array. Use the **search_n** function to search for two consecutive elements with value **2.4**.

23.7 Create an **array** with these numbers: **2.4**, **1.3**, **2.4**, **2.4**, **4.5**, **6.7**, **4.5**, **9.0**. Use the **sort** function to sort the array. Use the **bianry_search** function to search for value **6.7** and **4.3** respectively.

23.8 Implement the **fill** and **fill_n** functions.

```
template<typename ForwardIterator, typename T>
void fill(ForwardIterator beg, ForwardIterator end, const T& value)

template<typename ForwardIterator, typename size, typename T>
void fill_n(ForwardIterator beg, size n, const T& value)
```

23.9 Implement the **generate** and **generate_n** functions.

```
template<typename ForwardIterator, typename function>
void generate(ForwardIterator beg, ForwardIterator end, function gen)

template<typename ForwardIterator, typename size, typename function>
void generate_n(ForwardIterator beg, size n, function gen)
```

23.10 Implement the **reverse** and **reverse_copy** functions.

```
template<typename BidirectionalIterator>
void reverse(BidirectionalIterator beg,
  BidirectionalIterator end)
```

```
template<typename BidirectionalIterator, typename OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator beg,
  BidirectionalIterator end, OutputIterator targetPosition)
```

23.11 Implement the **replace** and **replace_if** functions.

```
template<typename ForwardIterator, typename T>
void replace(ForwardIterator beg, ForwardIterator end,
  const T& oldValue, const T& newValue)

template<typename ForwardIterator, typename boolFunction, typename T>
void replace_if(ForwardIterator beg, ForwardIterator end,
  boolFunction f, const T& newValue)
```

23.12 Implement the **find** and **find_if** functions.

```
template<typename InputIterator, typename T>
InputIterator find(InputIterator beg, InputIterator end, T& value)

template<typename InputIterator, typename boolFunction>
InputIterator find_if(InputIterator beg, InputIterator end,
  boolFunction f)
```