CHAPTER

20

STL Algorithms

Objectives

- To know how to use various types of iterators with the STL algorithms (§§20.1-20.20).
- To know the four types of STL algorithm (§20.2).
- To use the `copy` algorithm (§20.3).
- To use the algorithms `fill`, `fill_n`, `generate` and `generate_n` (§§20.4-20.5).
- To use the algorithms `remove`, `remove_if`, `remove_copy`, and `remove_copy_if` (§20.6).
- To use Boolean functions to specify criteria for STL algorithms (§20.6).
- To use the algorithms `replace`, `replace_if`, `replace_copy`, and `replace_copy_if`, `find`, `find_if`, `find_end`, and `find_first_of` (§§20.7-20.8).
- To use the algorithms `search`, `search_n`, `sort`, `binary_search`, `adjacent_find`, `merge`, and `inplace_merge` (§§20.9-20.11).
- To use function objects in STL algorithms (§20.10).
- To use the algorithms `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `swap`, `iter_swap`, and `swap_range` (§§20.12-20.14).
- To use the algorithms `count`, `count_if`, `max_element`, `min_element`, `random_shuffle`, `for_each`, and `transform` (§§20.15-20.18).
- To use set algorithms `includes`, `set_union`, `set_difference`, `set_intersection`, `set_symmetric_difference` (§20.19).
- To use numeric algorithms `accumulate`, `adjacent_sort`, `inner_product`, and `partial_sum` (§20.20).
20.1 Introduction

You have used the `find` function to find an element in an associative container, and the `sort` and `merge` functions to sort and merge lists. These functions are common to all containers. Rather than implementing them in each container, the STL supports these operations as generic algorithms that can be applied to a variety of containers as well as arrays. The algorithms operate on the elements through iterators.

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 generically applied to all containers through iterators. The STL makes the algorithms and containers easy to maintain.

NOTE:

*Side Remark: function and algorithm*

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

20.2 Types of Algorithms

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

*Side remark: nonmodifying algorithms*

- **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 20.1.

*Side remark: modifying algorithms*

- **Modifying Algorithms**: Modifying algorithms modifies the elements in the container by insertion, removing, rearranging, and changing values of the elements. Table 20.2 lists these algorithms.

*Side remark: numeric algorithms*

- **Numeric Algorithms**: Numeric algorithms provide four numeric operations for computing accumulate, adjacent difference, partial sum, and inner product. Table 20.3 lists these algorithms.

*Side remark: heap 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 20.1

Nonmodifying Algorithms

<table>
<thead>
<tr>
<th>adjacent_find</th>
<th>find</th>
<th>lower_bound</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_search</td>
<td>find_end</td>
<td>mismatch</td>
<td>search_n</td>
</tr>
<tr>
<td>count</td>
<td>find_first_of</td>
<td>max</td>
<td>upper_bound</td>
</tr>
<tr>
<td>count_if</td>
<td>find_if</td>
<td>max_element</td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>for_each</td>
<td>min</td>
<td></td>
</tr>
<tr>
<td>equal_range</td>
<td>includes</td>
<td>min_element</td>
<td></td>
</tr>
</tbody>
</table>

Table 20.2

Modifying Algorithms

<table>
<thead>
<tr>
<th>copy</th>
<th>prev_permutation</th>
<th>rotate_copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy_backward</td>
<td>random_shuffle</td>
<td>set_difference</td>
</tr>
<tr>
<td>fill</td>
<td>remove</td>
<td>set_intersection</td>
</tr>
<tr>
<td>fill_n</td>
<td>remove_copy</td>
<td>set_symmetric_difference</td>
</tr>
<tr>
<td>generate</td>
<td>remove_copy_if</td>
<td>set_union</td>
</tr>
<tr>
<td>generate_n</td>
<td>remove_if</td>
<td>sort</td>
</tr>
<tr>
<td>inplace_merge</td>
<td>replace</td>
<td>stable_partition</td>
</tr>
<tr>
<td>iter_swap</td>
<td>replace_copy</td>
<td>stable_sort</td>
</tr>
<tr>
<td>merge</td>
<td>replace_copy_if</td>
<td>swap</td>
</tr>
<tr>
<td>next_permutation</td>
<td>replace_if</td>
<td>swap_ranges</td>
</tr>
<tr>
<td>nth_element</td>
<td>reverse</td>
<td>transform</td>
</tr>
<tr>
<td>partial_sort</td>
<td>reverse_copy</td>
<td>unique</td>
</tr>
<tr>
<td>partial_sort_copy</td>
<td>rotate</td>
<td>unique_copy</td>
</tr>
<tr>
<td>partition</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 20.3

Numeric Algorithms

<table>
<thead>
<tr>
<th>accumulate</th>
<th>adjacent_difference</th>
<th>inner_product</th>
<th>partial_sum</th>
</tr>
</thead>
</table>

Table 20.4

Heap Algorithms

<table>
<thead>
<tr>
<th>make_heap</th>
<th>pop_heap</th>
<th>push_heap</th>
<th>sort_heap</th>
</tr>
</thead>
</table>

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 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 section gives the examples to demonstrate some frequently used algorithms.

20.3 copy

The copy function can be used to copy elements in a sequence from one container to another. The syntax is

```cpp
#include <algorithm>

template <typename inputIterator, typename outputIterator>
outputIterator copy(inputIterator beg, inputIterator end, outputIterator targetPostition)
```

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.

**NOTE:**

Many STL functions copy elements. These functions return an iterator that points to the next position past the last element copied.

Listing 20.1 demonstrates how to use the `copy` function.

```cpp
Listing 20.1 CopyDemo.cpp (Using copy Function )
***PD: Please add line numbers (including space lines) in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***
#include <algorithm>

// create a vector

// create a list

// copy to vector

// copy to list

// insert to list

// iterator for ostream

// copy to cout

// copy to list

// copy to cout
```
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <iterator>
using namespace std;

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

    copy(values + 2, values + 4, intVector.begin());
    copy(values, values + 5, intList.begin());

    cout << "After initial copy intVector: ";
    for (int i = 0; i < intVector.size(); i++)
    {
        cout << intVector[i] << " ";
    }

    cout << "\nAfter initial copy intList: ";
    for (list<int>::iterator p = intList.begin(); p != intList.end(); p++)
    {
        cout << "p << " ";
    }

    intVector.insert(intVector.begin(), 747);
    ostream_iterator<int> output(cout, " ");
    cout << "\nAfter the insertion function, intVector: ";
    copy(intVector.begin(), intVector.end(), output);

    cout << "\nAfter the copy function, intList: ";
    copy(intVector.begin(), intVector.begin() + 4, intList.begin());
    copy(intVector.begin(), intVector.end(), output);

    return 0;
}

<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
<end of output>

The program creates an array (line 10), a vector (line 11), and a list (line 12). The copy function copies the elements in the array values[2] and values[3] to the beginning of the vector (line 14), and the elements values[2], values[3], values[4], and values[5] 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 32), and copies the vector to the output stream iterator (line 33):

```
<Side Remark: copy to output>
    copy(intVector.begin(), intVector.end(), output);
```

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

TIP:

```
<Side Remark: write elements>
    It is convenient to use the copy function to write the elements from a container to an output stream.
```

CAUTION:

```
<Side Remark: possible copy errors>
    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.
```

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

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

***End CAUTION

20.4 fill and fill_n

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

```
<Side Remark: fill>
    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:

```
<Side Remark: fill n>
    template <typename forwardIterator, typename size, typename T>
        void fill_n(forwardIterator beg, size n, const T &value)
```
Listing 20.2 demonstrates how to use these two functions.

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

int main()
{
    int values[] = {1, 2, 3, 4, 5, 6};
    list<int> intList(values, values + 6);

    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 6, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);

    fill(values + 2, values + 4, 88);
    fill_n(intList.begin(), 2, 99);

    cout << "After the fill function, values: ";
    copy(values, values + 6, output);
    cout << "After the fill n function, intList: ";
    copy(intList.begin(), intList.end(), output);

    return 0;
}
```

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.
20.5 **generate and generate_n**

The functions `generate` and `generate_n` fill a sequence with a value returned from a function, using the following syntax:

```cpp
<Side Remark: generate>
template <typename forwardIterator, typename function>
void generate(forwardIterator beg, forwardIterator end, function gen);
</Side Remark: generate>

<Side Remark: generate_n>
template <typename forwardIterator, typename size, typename function>
void generate_n(forwardIterator beg, size n, function gen);
</Side Remark: generate_n>
```

Listing 20.3 demonstrates how to use these two functions.

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

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

int main()
{
    int values[] = {1, 2, 3, 4, 5, 6};
    list<int> intList(values, values + 6);
    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 6, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);
    generate(values + 2, values + 4, nextNum);
    generate_n(intList.begin(), 2, nextNum);

    return 0;
}
```

***PD: Please add line numbers (including space lines) in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

<Side Remark line 2: include algorithm>
<Side Remark line 7: function nextNum>
<Side Remark line 9: static local variable>
<Side Remark line 15: create an array>
<Side Remark line 16: create a list>
<Side Remark line 24: fill generated values>
<Side Remark line 25: fill generated values>
cout << "\nAfter the generate function, values: ";
copy(values, values + 6, output);
cout << "\nAfter the generate n function, intList: ";
copy(intList.begin(), intList.end(), output);
return 0;
}

The program creates an array (line 9) and a list (line 10). 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 the value of n 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.

20.6 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:

<Side Remark: remove>

```
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:

<Side Remark: remove_if>

```
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:

<Side Remark: remove_copy>

```
template <typename inputIterator, typename outputIterator, typename T>
```
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:

```cpp
outputIterator remove_copy(inputIterator beg, inputIterator end,
 outputIterator targetPosition, const T &value)
```

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

**NOTE:**

**<Side Remark: Boolean function>**

Many 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:**

**<Side Remark: size not changed>**

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:**

**<Side Remark: source not changed>**

The `remove_copy` and `remove_copy_if` functions copy the new contents to the target container, but do not change the source container.

Listing 20.4 demonstrates how to use the functions `remove` and `remove_if`.

---

**Listing 20.4** RemoveDemo.cpp (Removing Elements)

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

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***
```cpp
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

bool greaterThan3(int value)
{
    return value > 3;
}

int main()
{
    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
    list<int> intList(values, values + 8);

    // Output initial contents
    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 8, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);

    remove(values, values + 8, 3);
    remove_if(intList.begin(), intList.end(), greaterThan3);

    cout << "After the remove function, values: ";
    copy(values, values + 8, output);
    cout << "After the remove if function, intList: ";
    copy(intList.begin(), intList.end(), output);

    return 0;
}
```

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

The \texttt{remove if} function (line 24) removes all the elements in the list such that \texttt{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 20.5 demonstrates how to use the functions \texttt{remove\_copy} and \texttt{remove\_copy\_if}.

Listing 20.5 RemoveCopyDemo.cpp (Remove Copy Elements)

***PD: Please add line numbers (including space lines) in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***

\texttt{<Side Remark line 2: include algorithm>}
\texttt{<Side Remark line 7: function greaterThan3>}
\texttt{<Side Remark line 14: create an array>}
\texttt{<Side Remark line 15: create a list>}
\texttt{<Side Remark line 23: create an array>}
\texttt{<Side Remark line 24: create a list>}
\texttt{<Side Remark line 25: remove\_copy function>}
\texttt{<Side Remark line 26: remove\_copy\_if function>}

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

bool greaterThan3(int value)
{
    return value > 3;
}

int main()
{
    int values[] = \{1, 7, 3, 4, 3, 6, 1, 2\};
    list<int> intList(values, values + 8);

    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 8, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);

    int newValues[] = \{9, 9, 9, 9, 9, 9, 9, 9\};
    list<int> newIntList(values, values + 8);
    remove_copy(values, values + 8, newValues, 3);
    remove_copy_if(intList.begin(), intList.end(), newIntList.begin(), greaterThan3);

    cout << "After the remove_copy function, values: ";
    copy(values, values + 8, output);
    cout << "After the remove_copy_if function, intList: ";
    copy(intList.begin(), intList.end(), output);
}
```
cout << "\nAfter the remove_copy function, newValues: ";
copy(newValues, newValues + 8, output);
cout << "\nAfter the remove_copy if function, newIntList: ";
copy(newIntList.begin(), newIntList.end(), output);

return 0;

<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 function, intList: 1 7 3 4 3 6 1 2
After the remove_copy if function, newValues: 1 7 3 4 3 6 1 2
After the remove_copy if function, newIntList: 1 3 3 1 2 6 1 2
<end of output>

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, 9, 9}. After the copy, array newValues 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, 9, 9}. After the copy, list newIntList becomes {1, 3, 3, 1, 2, 6, 1, 2}.

20.7 replace, replace if, replace_copy, and replace_copy if

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

<Side Remark: replace>

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

The replace if function replaces all occurrence of the element for which boolFunction(element) is true, with a new value, using the following syntax:

<Side Remark: replace if>

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:
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:

Listing 20.6 demonstrates how to use the functions replace and replace_if.

// ReplaceDemo.cpp (Replacing Elements)

#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>

using namespace std;

bool greaterThan3(int value)
```cpp
int main()
{
    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
    list<int> intList(values, values + 8);

    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 8, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);

    replace(values, values + 8, 3, 747);
    replace_if(intList.begin(), intList.end(), greaterThan3, 747);

    cout << "After the replace function, values: ";
    copy(values, values + 8, output);
    cout << "After the replace if function, intList: ";
    copy(intList.begin(), intList.end(), output);

    return 0;
}
```

The program creates an array (line 14), a list (line 15), and display 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 20.7 demonstrates how to use the functions `replace_copy` and `replace_copy_if`.

**Side Remark line 2: include algorithm**

**Side Remark line 7: function `greaterThan3`**
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

bool greaterThan3(int value)
{
    return value > 3;
}

int main()
{
    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
    list<int> intList(values, values + 8);

    ostream_iterator<int> output(cout, " ");
    cout << "Initial contents, values: ";
    copy(values, values + 8, output);
    cout << "Initial contents, intList: ";
    copy(intList.begin(), intList.end(), output);

    int newValues[] = {9, 9, 9, 9, 9, 9, 9, 9};
    list<int> newIntList(values, values + 8);
    replace_copy(values + 2, values + 5, newValues, 3, 88);
    replace_copy_if(intList.begin(), intList.end(), newIntList.begin(), greaterThan3, 88);

    cout << "After the replace_copy function, values: ";
    copy(values, values + 8, output);
    cout << "After the replace_copy if function, intList: ";
    copy(intList.begin(), intList.end(), output);
    cout << "After the replace_copy function, newValues: ";
    copy(newValues, newValues + 8, output);
    cout << "After the replace_copy if function, newIntList: ";
    copy(newIntList.begin(), newIntList.end(), output);

    return 0;
}

<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 88 9 9 9 9
After the replace_copy if function, newIntList: 88 88 9 9 9 9
<end of output>

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 newValues 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}.

20.8 find, find_if, find_end, and find_first_of

The functions find, find_if, find_end, and find_first_of can be used to find the elements in sequence.

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

<Side Remark: find>

    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:

<Side Remark: find_if>

    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 20.8 demonstrates how to use the functions find and find_if.

Listing 20.8 FindDemo.cpp (Find Elements)

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

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

<Side Remark line 2: include algorithm>
<Side Remark line 9: create an array>
<Side Remark line 10: create a vector>
<Side Remark line 18: enter a key>
<Side Remark line 22: find a key>
<Side Remark line 23: found?>
<Side Remark line 24: position>
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    int values[] = {1, 7, 3, 4, 3, 6, 1, 2};
    vector<int> intVector(values, values + 8);
    ostream_iterator<int> output(cout, " ");
    cout << "values: ";
    copy(values, values + 8, output);
    cout << "intVector: ";
    copy(intVector.begin(), intVector.end(), output);
    int key;
    cout << "Enter a key: ";
    cin >> key;
    cout << "Find " << key << " in values: ";
    int *p = find(values, values + 8, key);
    if (p != values + 8)
        cout << "found at position " << (p - values);
    else
        cout << "not found";
    cout << "Find " << key << " in intVector: ";
    vector<int>::iterator itr = find(intVector.begin(), intVector.end(), key);
    if (itr != intVector.end())
        cout << "found at position " << (itr - intVector.begin());
    else
        cout << "not found";

    return 0;
}

<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
<end of output>

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 position of the matching element.
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 30). If found, `(itr - intVector.end())` is the position of the matching element.

**<Side Remark: find_end>**
The `find_end` function is to search a subsequence. It has two versions:

```cpp
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 20.9 demonstrates how to use the two versions of the `find_end` function.

---

**Listing 20.9 FindEndDemo.cpp (find_end Demo)**

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

int main()
```

---

725
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}. 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.

**<Side Remark: find first of>**
The function `find first of` searches the first common element in two sequences. It has two versions:

```
template <typename forwardIterator1, typename forwardIterator2>
forwardIterator1 find first of(forwardIterator1 beg1,
                        forwardIterator1 end1, forwardIterator2 beg2,
```


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

Listing 20.10 demonstrates how to use the two versions of the \texttt{find_first_of} function.

Listing 20.10 FindFirstOfDemo.cpp (find_first_of Demo)

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

bool greaterThan(int e1, int e2)
{
    return e1 > e2;
}

int main()
{
    int array1[] = {1, 7, 3, 4, 3, 6, 1, 2};
    int array2[] = {9, 96, 21, 3, 2, 3, 1};
    vector<int> intVector(array1, array1 + 8);
    
    ostream_iterator<int> output(cout, " ");
    
    
    return 0;
}
```
cout << "array1: ";
copy(array1, array1 + 8, output);
cout << "\nintVector: ";
copy(intVector.begin(), intVector.end(), output);

int *p = find_first_of(array1, array1 + 8, array2 + 2, array2 + 4);
if (p != array1 + 8)
    cout << "\nfind first of \{21, 3\} in array1 at position "
    << (p - array1);
else
    cout << "\nnot found";

vector<int>::iterator itr =
    find_first_of(intVector.begin(), intVector.end(),
        array2 + 2, array2 + 4, greaterThan);
if (itr != intVector.end())
    cout << "\nfind \{21, 3\} in intVector at position " <<
        (itr - intVector.begin());
else
    cout << "\nnot found";

return 0;

<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
<end of output>

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: {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 4 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.

20.9 search and search_n

The function search is similar to the function find_end. Both searches 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, return end1. In the first version, the elements are compared for equality; in the second version, the comparison boolFunction(elementInFirstSequence, elementInSecondSequence) must be true.

The search_n function searches for consecutive occurrence 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, return end. In the first version, the elements are compared for equality; in the second version, the comparison boolFunction(element) must be true.

Listing 20.11 demonstrates how to use the functions search and search_n.
```cpp
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    int array1[] = {1, 7, 3, 4, 3, 1, 2};
    int array2[] = {9, 96, 4, 3, 2, 3, 1};
    vector<int> intVector(array1, array1 + 8);

    ostream_iterator<int> output(cout, " ");
    cout << "array1: " << array1 << endl;
    copy(array1, array1 + 8, output);
    cout << "intVector: " << intVector << endl;
    copy(intVector.begin(), intVector.end(), output);

    int *p = search(array1, array1 + 8, array2 + 2, array2 + 4);
    if (p != array1 + 8)
    {
        cout << "Search {4, 3} in array1 at position " << (p - array1) << endl;
    }
    else
    {
        cout << "not found" << endl;
    }

    vector<int>::iterator itr = search_n(intVector.begin(), intVector.end(), 2, 3);
    if (itr != intVector.end())
    {
        cout << "Search two occurrence of 3 in intVector at position " << (itr - intVector.begin()) << endl;
    }
    else
    {
        cout << "not found" << endl;
    }

    return 0;
}
```

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

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

<output>
array1: 1 7 3 4 3 1 2
intVector: 1 7 3 4 3 3 1 2
Search {4, 3} in array1 at position 3
Search two occurrence of 3 in intVector at position 4
<end of output>
Invoking search n(intVector.begin(), intVector.end(), 2, 3) searches for two consecutive 3. The matching position is 4 in array1.

20.10 sort and 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:

**<Side Remark: sort>**

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

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

The binary search function searches a value in a sorted sequence, using one of the two versions:

**<Side Remark: binary_search>**

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

```cpp
template <typename forwardIterator, typename T, 
          typename relationOperator>
bool binary_search(forwardIterator beg, 
                   forwardIterator end, const T &value, relationOperator op)
```

NOTE:

**<Side Remark: function object>**

Many STL algorithms allow you to pass a function operator. It is actually a function object and 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 20.5. The sort and binary_search algorithm require the relational operator. To use function objects, include the `<functional>` header.

Table 20.5

Function Objects
### STL function object vs. type

<table>
<thead>
<tr>
<th>STL function object</th>
<th>Type</th>
<th>STL function object</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equal_to&lt;T&gt;</code></td>
<td>relational</td>
<td><code>plus&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>not_equal_to&lt;T&gt;</code></td>
<td>relational</td>
<td><code>minus&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>greater&lt;T&gt;</code></td>
<td>relational</td>
<td><code>multiplies&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>greater_equal&lt;T&gt;</code></td>
<td>relational</td>
<td><code>divides&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>less&lt;T&gt;</code></td>
<td>relational</td>
<td><code>modulus&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>less_equal&lt;T&gt;</code></td>
<td>relational</td>
<td><code>negate&lt;T&gt;</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>logical_and&lt;T&gt;</code></td>
<td>logical</td>
<td><code>logical_not&lt;T&gt;</code></td>
<td>logical</td>
</tr>
<tr>
<td><code>logical_or&lt;T&gt;</code></td>
<td>logical</td>
<td><code>logical_or&lt;T&gt;</code></td>
<td>logical</td>
</tr>
</tbody>
</table>

Listing 20.12 demonstrates how to use the functions `sort` and `binary_search`.

Listing 20.12 SortDemo.cpp (Sort Elements)

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

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

<Side Remark line 2: include algorithm>
<Side Remark line 8: create an array>
<Side Remark line 14: sort array>
<Side Remark line 19: binary search>

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

int main()
{
    int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};

    ostream_iterator<int> output(cout, " ");
    cout << "Before sort, array1: ";
    copy(array1, array1 + 8, output);
    sort(array1, array1 + 8);
    cout << "After sort, array1: ";
    copy(array1, array1 + 8, output);
    cout << (binary_search(array1, array1 + 8, 4) ? "4 is in array1" : "4 is not in array1");
    sort(array1, array1 + 8, greater_equal<int>());
    cout << "After sort, array1: ";
    copy(array1, array1 + 8, output);
    cout << (binary_search(array1, array1 + 8, 4) ? "\n4 is in array1" : "\n4 is not in array1");

    cout << "\nAfter sort, array1: ";
    copy(array1, array1 + 8, output);
    cout << (binary_search(array1, array1 + 8, 4) ? "\n4 is in array1" : "\n4 is not in array1");
```

732
greater_equal<int>() ?
   "\n4 is in array1" : "\n4 is not in array1"
};

return 0;

<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 not in array1
<end of output>

The default function operator for \texttt{sort} and \texttt{binary\_search} is \texttt{less\_equal<T>()}. Invoking \texttt{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 have invoking \texttt{binary\_search(array1, array1 + 8, 4, greater\_equal<int>());} (lines 28-29) to perform binary search.

\textbf{20.11 adjacent\_find, merge, and inplace\_merge}

The \texttt{adjacent\_find} function looks for first occurrence of adjacent elements of equal value or satisfying \texttt{boolFunction(element), using the following syntax:}

\begin{verbatim}<Side Remark: adjacent\_find>
template <typename forwardIterator>
forwardIterator adjacent_find(forwardIterator beg,
                             forwardIterator end)

template <typename forwardIterator, typename boolFunction>
forwardIterator adjacent_find(forwardIterator beg,
                             forwardIterator end, boolFunction f)
<end of remark>
\end{verbatim}

The \texttt{adjacent\_find} function returns the iterator that points to the first element in the matching sequence. If not found, it returns \texttt{end}.

The \texttt{merge} function merges two sorted sequences into a new sequence, using the following syntax:

\begin{verbatim}<Side Remark: merge>
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>
\end{verbatim}
The \texttt{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:

\begin{verbatim}
<Side Remark: \textit{inplace\_merge}>
  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)
\end{verbatim}

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

Listing 20.13 demonstrates how to use the functions \texttt{adjacent\_find}, \texttt{merge}, and \texttt{inplace\_merge}.

\begin{verbatim}
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

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

    // List is already sorted, so merge and inplace_merge
    // should not change it.
    cout << "intList: ";
    ostream_iterator<int> output(cout, " ");
    for (list<int>::iterator it = intList.begin(); it != intList.end(); ++it)
        cout << *it << " ";
    cout << endl;

    // Sort the first half of array1
    cout << "Sorted Array1: ";
    for (int i = 0; i < 7; i++)
        cout << array1[i] << " ";
    cout << endl;

    // Merge arrays
    cout << "Array1 and array2 merged: ";
    for (int i = 0; i < 7; i++)
        cout << array1[i] + array2[i] << " ";
    cout << endl;

    // In-place merge
    cout << "In-place Merge: ";
    for (int i = 0; i < 9; i++)
        cout << array1[i] + array2[i] << " ";
    cout << endl;
}
\end{verbatim}
The program creates an array and a list (lines 9-10). The contents of the array is:

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}

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

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

20.12 reverse and reverse copy

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 are:

.Side Remark: reverse>
template <typename bidirectionalIterator>
void reverse(bidirectionalIterator beg,
            bidirectionalIterator end)

.Side Remark: reverse_copy

template <typename bidirectionalIterator, typename outputIterator>
outputIterator reverse_copy(bidirectionalIterator beg,
                            bidirectionalIterator end, outputIterator targetPosition)

Listing 20.14 demonstrates how to use the functions reverse, and reverse_copy.

Listing 20.14 ReverseDemo.cpp (Reverse Elements)

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

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

.Side Remark line 2: include algorithm
.Side Remark line 9: create an array
.Side Remark line 10: create a list
.Side Remark line 16: reverse array
.Side Remark line 17: reverse copy

cpp
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{
  int array1[] = {1, 7, 3, 4, 3, 3, 1, 2};
  list<int> intList(8);

  ostream_iterator<int> output(cout, " ");
  cout << "array1: ";
  copy(array1, array1 + 8, output);

  reverse(array1, array1 + 8);
  cout << "after reverse arrays, array1: ";
  copy(array1, array1 + 8, output);

  reverse_copy(array1, array1 + 8, intList.begin());
  cout << "after reverse_copy, array1: ";
  copy(array1, array1 + 8, output);

  cout << "after reverse_copy, intList: ";
  copy(intList.begin(), intList.end(), output);

  return 0;
}

OUTPUT
array1: 1 7 3 4 3 3 1 2
after reverse arrays, array1: 2 1 3 3 4 3 7 1
20.13 rotate and rotate_copy

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

 `<Side Remark: rotate>`

    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:

 `<Side Remark: rotate_copy>`

    template <typename forwardIterator, typename outputIterator>
    outputIterator rotate_copy(forwardIterator beg, forwardIterator newBeg, 
                                forwardIterator end, outputIterator targetPosition)

Listing 20.15 demonstrates how to use the functions reverse, and reverse_copy.

    Listing 20.15 RotateDemo.cpp (Rotate Elements)

    ***PD: Please add line numbers (including space lines) in the following code***
    ***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.***
    <Side Remark line 2: include algorithm>
    <Side Remark line 9: create an array>
    <Side Remark line 10: create a list>
    <Side Remark line 16: rotate array>
    <Side Remark line 20: rotate copy>

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

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

rotate_copy(array1, array1 + 1, array1 + 8, intList.begin());
cout << "\n after rotate_copy, array1: ";
copy(array1, array1 + 8, output);
cout << "\n after rotate_copy, intList: ";
copy(intList.begin(), intList.end(), output);

return 0;
}

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}

20.14 swap, iter swap, and swap ranges

The functions swap, iter swap, and swap range are used to swap elements. They are defined as follows:

<Side Remark: swap>
template <typename T>
void swap(T &value1, T &value2)

<Side Remark: iter swap>
template <typename forwardIterator1, typename forwardIterator2>
void iter_swap(forwardIterator1 p1, forwardIterator p2)

<Side Remark: swap range>
template <typename forwardIterator1, typename forwardIterator2>
forwardIterator swap_ranges(
    forwardIterator1 beg1, forwardIterator1 end1,
The \texttt{swap} function swaps the values in two variables. The \texttt{iter\_swap} function swaps the values pointed by the iterators. The \texttt{swap\_ranges} function swaps two sequences.

Listing 20.16 demonstrates how to use these three functions.

```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);

    cout << "\nafter swap ranges, array1: ";
    swap_ranges(array1, array1 + 4, array1 + 4);
    copy(array1, array1 + 8, output);

    return 0;
}
```

```
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
<end of output>
```
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).

20.15 **count and count_if**

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 20.17 demonstrates how to use these functions.

Listing 20.17 CountDemo.cpp (Count Elements)

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

bool greaterThan1(int value)
{
    return value > 1;
}

int main()
{
```

740
int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};

    cout << "The number of 1's in array1: " <<
            count(array1, array1 + 8, 1) << endl;

    cout << "The number of elements > 1 in array1: " <<
            count_if(array1, array1 + 8, greaterThan1) << endl;

    return 0;
}

<output>
The number of 1's in array1: 1
The number of elements > 1 in array1: 7
<end of output>

20.16 max element and min element

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:

<Side Remark: max element>
template <typename forwardIterator>
forwardIterator max_element(forwardIterator beg,
                          forwardIterator end)

<Side Remark: min element>
template <typename forwardIterator>
forwardIterator min_element(forwardIterator beg,
                          forwardIterator end)

Listing 20.18 gives an example of how to use these functions.
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

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

    return 0;
}

<output>
After random shuffle, array1: 2 6 4 3 7 5 1 8

20.17 random_shuffle

The random_shuffle function randomly reorders the elements in a sequence, using the following syntax:

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

Listing 20.19 gives an example of how to use this function.

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

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

    return 0;
}
```

<output>
After random shuffle, array1: 2 6 4 3 7 5 1 8
20.18 **for_each** and **transform**

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

<Side Remark: *for_each*>

```cpp
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:

<Side Remark: *transform*>

```cpp
template <typename inputIterator, typename outputIterator, typename function>
outputIteration transform(inputIterator beg, inputIterator end, outputIterator targetPosition, function f)
```

Listing 20.20 demonstrates how to use these functions.

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

void display(int &value)
{
    cout << value << " ";
}

int square(int &value)
{
    return value * value;
}

int main()
```
int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
cout << "array1: ";
for (array1, array1 + 8, display);

vector<int> intVector(8);
transform(array1, array1 + 8, intVector.begin(), square);
cout << "intVector: ";
for (intVector.begin(), intVector.end(), display);

return 0;

<output>
array1: 1 2 3 4 5 6 7 8
intVector: 1 4 9 16 25 36 49 64
<end of output>

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.

20.19 includes, set union, set difference, set intersection, and set symmetric difference

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 are already sorted.

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

<Side Remark: includes>

    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.

<Side Remark: set union>

    template <typename inputIterator1, typename inputIterator2,
              typename outputIterator>
    outputIterator set union(inputIterator1 beg1, inputIterator1 end1,
                              inputIterator2 beg2, inputIterator2 end2, outputIterator result);
The `set_difference` function obtains the elements that belong to the first sequence, but not in the second sequence.

```
<Side Remark: set_difference>
    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.

```
<Side Remark: set_intersection>
    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.

```
<Side Remark: set_intersection>
    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:

```
<table>
<thead>
<tr>
<th>Operation</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>array1 union array2</td>
<td>{1, 2, 3, 4, 5, 6, 7, 8, 9, 12}</td>
</tr>
<tr>
<td>array1 difference array2</td>
<td>{2, 4, 5, 7, 8}</td>
</tr>
<tr>
<td>array1 intersection array2</td>
<td>{1, 3, 6}</td>
</tr>
<tr>
<td>array1 symmetric_diff array2</td>
<td>{2, 4, 5, 7, 8, 9, 12}</td>
</tr>
</tbody>
</table>
```

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

Listing 20.21 demonstrates how to use set functions.

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

int main()
{
    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int array2[] = {1, 3, 6, 9, 12};
    vector<int> intVector(15);

    ostream_iterator<int> output(cout, " ");
    copy(array1, array1 + 8, output);
    cout << "array1: 
array2: 
bool isContained =
    includes(array1, array1 + 8, array2, array2 + 3);
    cout << (isContained ? "{1, 3, 6} is a subset of array1" : "\n{1, 3, 6} is not a subset of array1");
    vector<int>::iterator last = set_union(array1, array1 + 8, array2, array2 + 5, intVector.begin());
    cout << "\nAfter union, intVector: ";
    copy(intVector.begin(), last, output);
    last = set_difference(array1, array1 + 8, array2, array2 + 5, intVector.begin());
    cout << "\nAfter difference, intVector: ";
```

---

746
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

\[
\text{intVector} \rightarrow \{2, 4, 5, 7, 8, 9, 12\}
\]

NOTE:

**<Side Remark: resulting container>**
The set operations store the contents to a result container. The size of this result container 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().

### 20.20 accumulate, adjacent_difference, inner_product, and partial_sum

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.

**<Side Remark: accumulate>**

```
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,

<table>
<thead>
<tr>
<th>array1</th>
<th>result of accumulate(array1, array1 + 5, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array1</th>
<th>result of accumulate(array1, array1 + 5, 1, multiplies&lt;int&gt;())</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>120</td>
</tr>
</tbody>
</table>

The adjacent_difference function has two versions.

**<Side Remark: adjacent_difference>**

```
template <typename inputIterator, typename T>
outputIterator adjacent_difference(inputIterator beg,
```
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 element. The second version is the same as the first version except that the specified arithmetic operator is applied to replace the subtraction operator. For example,

<table>
<thead>
<tr>
<th>array1</th>
<th>result of adjacent_difference(array1, array1 + 5, intVector.begin())</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>{1, 1, 1, 1, 1}</td>
</tr>
</tbody>
</table>

The **inner_product** function has two versions.

**<Side Remark: inner_product>**

```cpp
template <typename inputIterator1, typename inputIterator2, 
    typename T>
T inner_product(inputIterator1 beg1,
    inputIterator1 end1, inputIterator2 beg2, T initValue)
```

```cpp
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 \{a_1, a_2, ..., a_i\} and \{b_1, b_2, ..., b_i\} is defined as

\[ a_1 * b_1 + a_2 * b_2 + ... + a_i * b_i \]

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,

<table>
<thead>
<tr>
<th>array1</th>
<th>result of inner_product(array1, array1 + 5, array1, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>55</td>
</tr>
</tbody>
</table>

The **partial_sum** function has two versions.

**<Side Remark: partial_sum>**
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,

<table>
<thead>
<tr>
<th>array1</th>
<th>result of</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>{1, 3, 6, 10, 15}</td>
</tr>
</tbody>
</table>

Listing 20.22 demonstrates how to use the mathematical functions.

***PD: Please add line numbers (including space lines) in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks.
<Side Remark line 2: include algorithm>
<Side Remark line 3: include numeric>
<Side Remark line 6: include functional>
<Side Remark line 11: create array1>
<Side Remark line 12: create intVector>
<Side Remark line 19: accumulate>
<Side Remark line 22: accumulate>
<Side Remark line 25: adjacent_difference>
<Side Remark line 30: inner_product>
<Side Remark line 32: partial_sum>

```cpp
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int array1[] = {1, 2, 3, 4, 5};
    vector<int> intVector(5);
```
```cpp
    _ostream_iterator<int> output(cout, " ");
    cout << "array1: ";
    copy(array1, array1 + 5, output);
    cout << "\nSum of array1: " <<
         accumulate(array1, array1 + 5, 0) << endl;
    cout << "Product of array1: " <<
         accumulate(array1, array1 + 5, 1, multiplies<int>()) << endl;
    vector<int>::iterator last =
        adjacent_difference(array1, array1 + 5, intVector.begin());
    cout << "After adjacent difference, intVector: ";
    copy(intVector.begin(), last, output);
    cout << "\nInner product of array1 * array1 is " <<
         inner_product(array1, array1 + 5, array1, 0);
    last = partial_sum(array1, array1 + 5, intVector.begin());
    cout << "\nAfter partial sum, intVector: ";
    copy(intVector.begin(), last, output);
    return 0;
```

```
<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
<end of output>
```

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

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

Invoking `accumulate(array1, array1 + 5, 0)` (line 18) returns the sum of all the elements in `array1`.

Invoking `accumulate(array1, array1 + 5, 1, multiplies<int>())` (line 21) returns the multiplication of all the all the elements in `array1`.

Invoking `adjacent_difference(array1, array1 + 5, intVector.begin())` (line 24) obtains a sequence for the adjacent difference of `array1` in `intVector`.

Invoking `inner_product(array1, array1 + 5, array1, 0)` (line 29) obtains the inner product of `array1` and `array1`.

Invoking `partial_sum(array1, array1 + 5, intVector.begin())` (line 31) obtains a sequence for the partial sum of `array1` in `intVector`. 751
Key Terms

***PD: Please place terms in two columns same as in intro5e.

- accumulate algorithm
- adjacent_find algorithm
- adjacent_sort algorithm
- binary_search algorithm
- copy algorithm
- count algorithm
- count_if algorithm
- fill algorithm
- fill_in 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
Chapter Summary

- The STL separates algorithms from the containers. This enables the algorithms to be generically applied to all containers through iterators. The STL makes the algorithms and containers easy to maintain.
- The STL provides approximately 80 algorithms. The algorithms can be classified into four groups: nonmodifying algorithms, modifying algorithms, numeric algorithms, and heap algorithms.
- All the algorithms operate through iterators. 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 copy function can be used to copy elements in a sequence from one container to another.
- The functions fill and fill_n can be used to fill a container with a specified value.
- The functions generate and generate_n fill a container with a value returned from a function.
- The functions remove, remove_if, remove_copy, and remove_copy_if remove the elements from a sequence that matches some criteria.
- The functions replace, replace_if, replace_copy, and replace_copy_if replace all occurrence of a given value with a new value in a sequence.
- The functions find, find_if, find_end, and find_first_of can be used to find the elements in sequence.
- The functions search and search_n search for a subsequence.
- The sort function requires random-access iterators. You can apply it to sort an array, vector, or deque.
• The `adjacent_find` function looks for first occurrence of adjacent elements of equal value or satisfying `boolFunction(element)`.
• The `merge` function merges two sorted sequences into a new sequence.
• 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 `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 `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.
• The `swap` function swaps the values in two variables. The `iter_swap` function swaps the values pointed by the iterators. The `swap_range` function swaps two sequences.
• 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.
• The functions `max_element` and `min_element` obtain the maximum element and minimum element in a sequence.
• The `random_shuffle` function randomly reorders the elements in a sequence.
• The `for_each` function is used to process each element in a sequence by applying a function. The `transform` function is to apply a function on each element in the sequence and copy the result to a target sequence.
• 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 are already sorted.
• The STL supports the mathematical functions `accumulate`, `adjacent_difference`, `inner_product`, and `partial_sum`. They are defined in the `<numeric>` header.

Review Questions

Sections 20.1-20.2

20.1

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

20.2
What are the four types of STL algorithms?

Section 20.3 **copy**

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

```cpp
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);
```

Section 20.4 **fill** and **fill_n**

What is wrong in the following code?

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

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

Section 20.5 **generate** and **generate_n**

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

```cpp
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);
```

Section 20.6 **nextNum**

What are the **nextNum** algorithms for? Show the printout of the following code:

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

Section 20.6 remove, remove_if, remove_copy, and remove_copy_if

20.7

What are the remove, remove_if, remove_copy, and remove_copy_if algorithms for? Show the printout of the following code:

```cpp
    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;
    }

Section 20.7 replace, replace_if, replace_copy, and replace_copy_if

20.8

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

```cpp
    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;
}

Section 20.8 find, find_if, find_end, and find_first_of

20.9

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

Section 20.9 search and search_n

20.10

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

Section 20.10 sort and binary_search

20.11

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?

Section 20.11 adjacent_find, merge, and inplace_merge

20.12

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);

Section 20.12 reverse and reverse_copy

20.13
What are the reverse and reverse_copy algorithms for? Does the reverse_copy algorithm change the contents of the original sequence?

Section 20.13 rotate and rotate_copy

20.14

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

```cpp
int values[] = {1, 2, 3, 4, 5, 1, 1};
rotate(values, values + 5, values + 8);

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

Section 20.14 swap, iter_swap, and swap_ranges

20.15

What are the swap, iter_swap, and swap_ranges algorithms for?

Section 20.15 count and count_if

20.16

What are the count and count_if algorithms for?

Section 20.16 max_element and min_element

20.17

What are the max and max_element algorithms for?

Section 20.17 random_shuffle

20.18

What is the random_shuffle algorithm for?

Section 20.18 for_each and transform

20.19

What are the for_each and transforms algorithms for?

Section 20.20 includes, set union, set difference, set intersection, and set symmetric difference

20.20
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.

Section 20.21 accumulate, adjacent_difference, inner_product, and partial_sum

20.21

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?

Programming Exercises

20.1

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.

20.2

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.

20.3

Create an array with 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.

20.4

Create an array with 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 by 9.9. Use the replace if function to replace all the elements that are less than 2.0 by 12.5. Use the replace_copy function to replace all occurrences of 6.7 by 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 by 747 and copy the sequence to a multiset.

20.5

Create an array with 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}.

20.6

Create an array with 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.

20.7

Create an array with 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.

20.8

Implement the fill and fill_n functions.

```cpp
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)
```

20.9

Implement the generate and generate_n functions.

```cpp
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)

20.10
Implement the remove and remove if functions.

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

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

20.11
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)

20.12
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)