Chapter 16
Standard Library Algorithms

C++ How to Program, 9/e

OBJECTIVES

In this chapter you'll:

- Program with many of the dozens of Standard Library algorithms.
- Use iterators with algorithms to access and manipulate the elements of Standard Library containers.
- Pass function pointers, function objects and lambda expressions into Standard Library algorithms.

16.1 Introduction

- This chapter continues our discussion of the Standard Library's containers, iterators and algorithms by focusing on algorithms that perform common data manipulations such as searching, sorting and comparing elements or entire containers.
- The Standard Library provides over 90 algorithms, many of which are new in C++11.
- Most of them use iterators to access container elements.
- As you'll see, various algorithms can receive a function pointer (a pointer to a function's code) as an argument.
- Such algorithms use the pointer to call the function—typically with one or two container elements as arguments.

16.2 Minimum Iterator Requirements

- *With few exceptions, the Standard Library separates algorithms from containers.*
- An important part of every container is the type of iterator it supports (Fig. 15.7).
- This determines which algorithms can be applied to the container.
- For example, both vectors and arrays support random-access iterators that provide all of the iterator operations shown in Fig. 15.9.
- All Standard Library algorithms can operate on vectors and the ones that do not modify a container's size can also operate on arrays.

16.2 Minimum Iterator Requirements (cont.)

- Each Standard Library algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality.
- If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that supports *forward iterators*, *bidirectional iterators* or *random-access iterators*.

Software Engineering Observation 16.1

Standard Library algorithms do not depend on the implementation details of the containers on which they operate. As long as a container's (or built-in array's) iterators satisfy the requirements of an algorithm, the algorithm can work on the container.

Portability Tip 16.1

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

16.2 Minimum Iterator Requirements (cont.)

Iterator Invalidation

- Iterators simply *point* to container elements, so it's possible for iterators to become *invalid* when certain container modifications occur.
- For example, if you invoke clear on a `vector`, *all* of its elements are *removed*.
- If a program had any iterators that pointed to that `vector`'s elements before `clear` was called, those iterators would now be *invalid*.

16.2 Minimum Iterator Requirements (cont.)

- Here we summarize when iterators are invalidated during *insert* and *erase* operations.
- When *inserting* into a:
  - `vector`—If the vector is reallocated, all iterators pointing to that `vector` are invalidated. Otherwise, iterators from the insertion point to the end of the `vector` are invalidated.
  - `deque`—All iterators are invalidated.
  - `list` or `forward_list`—All iterators *remain valid*.
  - Ordered associative container—All iterators *remain valid*.
  - Unordered associative container—All iterators are invalidated if the containers need to be reallocated.

16.2 Minimum Iterator Requirements (cont.)

- When *erasing* from a container, iterators to the *erased* elements are invalidated. In addition:
  - `vector`—Iterators from the erased element to the end of the `vector` are invalidated.
  - `deque`—If an element in the middle of the `deque` is erased, all iterators are invalidated.

16.3.3 remove, remove_if, remove_copy and remove_copy_if

- Figure 16.3 demonstrates removing values from a sequence with algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`.


```cpp
// Fig. 16.3: fig16_03.cpp
// Algorithms remove, remove_if, remove_copy and remove_copy_if.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <array> // array class-template definition
#include <iterator> // ostream_iterator
using namespace std;

bool greater9( int ); // prototype

int main()
{
   const size_t SIZE = 10;
   array< int, SIZE > init = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   array< int, SIZE > a1( init ); // initialize with copy of init
   cout << "a1 before removing all 10s:\n   ";
   copy( a1.cbegin(), a1.cend(), output );

```

Fig. 16.3 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 1 of 4.)


```cpp
   // remove all 10s from a1
   auto newLastElement = remove( a1.begin(), a1.end(), 10 );
   cout << "\a1 after removing all 10s:\n   ";
   copy( a1.begin(), newLastElement, output );

   array< int, SIZE > a2( init ); // initialize with copy of init
   array< int, SIZE > c = { 0 }; // initialize to 0s
   cout << "\n\a2 before removing all 10s and copying:\n   ";
   copy( a2.cbegin(), a2.cend(), output );

   // copy from a2 to c, removing 10s in the process
   remove_copy( a2.cbegin(), a2.cend(), c.begin(), 10 );
   cout << "\nc after removing all 10s from a2:\n   ";
   copy( c.cbegin(), c.cend(), output );

   array< int, SIZE > a3( init ); // initialize with copy of init
   cout << "\n\na3 before removing all elements greater than 9:\n   ";
   copy( a3.cbegin(), a3.cend(), output );

   // remove elements greater than 9 from a3
   newLastElement = remove_if( a3.begin(), a3.end(), greater9 );
   cout << "\na3 after removing all elements greater than 9:\n   ";
   copy( a3.begin(), newLastElement, output );
```

Fig. 16.3 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 2 of 4.)


```cpp

   array< int, SIZE > a4( init ); // initialize with copy of init
   array< int, SIZE > c2 = { 0 }; // initialize to 0s
   cout << "\n\na4 before removing all elements"
      << "\ngreater than 9 and copying:\n   ";
   copy( a4.cbegin(), a4.cend(), output );

   // copy elements from a4 to c2, removing elements greater
   // than 9 in the process
   remove_copy_if( a4.cbegin(), a4.cend(), c2.begin(), greater9 );
   cout << "\nc2 after removing all elements"
      << "\ngreater than 9 from a4:\n   ";
   copy( c2.cbegin(), c2.cend(), output );
   cout << endl;
} // end main

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

Fig. 16.3 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 3 of 4.)

```
a1 before removing all 10s:
   10 2 10 4 16 6 14 8 12 10
a1 after removing all 10s:
   2 4 16 6 14 8 12

a2 before removing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
c after removing all 10s from a2:
   2 4 16 6 14 8 12 0 0 0

a3 before removing all elements greater than 9:
   10 2 10 4 16 6 14 8 12 10
a3 after removing all elements greater than 9:
   2 4 6 8

a4 before removing all elements
greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
c2 after removing all elements
greater than 9 from a4:
   2 4 6 8 0 0 0 0 0 0
```

Fig. 16.3 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 4 of 4.)

16.3.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (Cont.)

***`remove`* Algorithm**

- Line 22 uses the remove algorithm to eliminate from `a1` *all* elements with the value `10` in the range from `a1.begin()` up to, but *not* including, `a1.end()`.
- The first two iterator arguments must be *forward* iterators.
- This algorithm does *not* modify the number of elements in the container or destroy the eliminated elements, but it does move *all* elements that are *not* eliminated toward the *beginning* of the container.
- The algorithm returns an iterator positioned after the last element that was not removed.
- *Elements from the iterator position to the end of the container have unspecified values.*

16.3.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (Cont.)

***`remove_if`* Algorithm**

- Line 32 uses the remove_copy algorithm to copy *all* elements from `a2` that do *not* have the value `10` in the range from `a2.cbegin()` up to, but *not* including, `a2.cend()`.
- The elements are placed in `c`, starting at position `c.begin()`.
- The iterators supplied as the first two arguments must be *input iterators*.
- The iterator supplied as the third argument must be an output iterator so that the element being copied can be *inserted* into the copy location.
- This algorithm returns an iterator positioned after the last element copied into `vector c`.

16.3.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (Cont.)

remove_if Algorithm

- Line 41 uses the remove_if algorithm to delete from `a3` *all* those elements in the range from `a3.begin()` up to, but *not* including, `a3.end()` for which our user-defined unary predicate function `greater9` returns `true`.
- Function `greater9` (defined in lines 61–64) returns `true` if the value passed to it's greater than 9; otherwise, it returns `false`.

16.3.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (Cont.)

- The iterators supplied as the first two arguments must be *forward* iterators.
- This algorithm does *not* modify the number of elements in the container, but it does move to the *beginning* of the container *all* elements that are *not* removed.
- This algorithm returns an iterator positioned after the last element that was *not* removed.
- All elements from the iterator position to the end of the container have *undefined* values.

16.3.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (Cont.)

***`remove_copy_if`* Algorithm**

- Line 53 uses the remove_copy_if algorithm to copy all those elements from `a4` in the range from `a4.cbegin()` up to, but *not* including, `a4.cend()` for which the *unary predicate function* `greater9` returns `true`.
- The elements are placed in `c2`, starting at `c2.begin()`.
- The iterators supplied as the first two arguments must be *input iterators*.
- The iterator supplied as the third argument must be an *output iterator* so that the element being copied can be *assigned* to the copy location.
- This algorithm returns an iterator positioned after the *last* element copied into `c2`.

Follow remove-like algorithms with erase if you really want to remove something

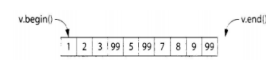
```
vector<int> v; // create a vector<int> and fill it with
v.reserve(10); //the values 1-10.
for (int i = 1; i <= 10; ++i)
{
      v.push_back(i);
}
cout << v.size(); //prints 10

v[3] = v[5] = v[9] = 99; // set 3 elements to 99

remove(v.begin(), v.end(), 99); // remove all elements with value 99
cout << v.size();               // still prints 10!
```

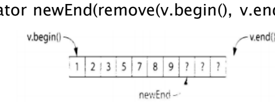

Follow remove-like algorithms with erase if you really want to remove something

prior to calling remove:

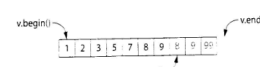


after calling

vector::iterator newEnd(remove(v.begin(), v.end(), 99) ; :



In almost all implementations it looks like this:



In general, after calling remove, the values removed from the range may or may not continue to exist in the range!

When using containers of newed pointers, remember to delete the pointers before the container is destroyed.

```
void doSomething()
{
    vector<Widget*> vwp;
    for (int i = 0; i < SOME_MAGIC_NUMBER; ++i)
          vwp.push_back(new Widget);
    … // use vwp
} //Widgets are leaked here!

→ delete elements before out-of-scope or use
shared_ptr's.
```


16.3.5 Mathematical Algorithms

- Figure 16.5 demonstrates several common mathematical algorithms, including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `minmax_element`, `for_each` and `transform`.


```cpp
// Fig. 16.5: fig16_05.cpp
// Mathematical algorithms of the Standard Library.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <numeric> // accumulate is defined here
#include <array>
#include <iterator>
using namespace std;

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

int main()
{
   const size_t SIZE = 10;
   array< int, SIZE > a1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   ostream_iterator< int > output( cout, " " );

   cout << "a1 before random_shuffle: ";
   copy( a1.cbegin(), a1.cend(), output );

```

Fig. 16.5 | Mathematical algorithms of the Standard Library. (Part 1 of 5.)


```cpp
   random_shuffle( a1.begin(), a1.end() ); // shuffle elements of a1
   cout << "\na1 after random_shuffle: ";
   copy( a1.cbegin(), a1.cend(), output );

   array< int, SIZE > a2 = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   cout << "\n\na2 contains: ";
   copy( a2.cbegin(), a2.cend(), output );

   // count number of elements in a2 with value 8
   int result = count( a2.cbegin(), a2.cend(), 8 );
   cout << "\nNumber of elements matching 8: " << result;

   // count number of elements in a2 that are greater than 9
   result = count_if( a2.cbegin(), a2.cend(), greater9 );
   cout << "\nNumber of elements greater than 9: " << result;

   // locate minimum element in a2
   cout << "\n\nMinimum element in a2 is: "
      << *( min_element( a2.cbegin(), a2.cend() ) );

   // locate maximum element in a2
   cout << "\nMaximum element in a2 is: "
      << *( max_element( a2.cbegin(), a2.cend() ) );

```

Fig. 16.5 | Mathematical algorithms of the Standard Library. (Part 2 of 5.)

```
47    // locate minimum and maximum elements in a2
48    auto minAndMax = minmax_element( a2.cbegin(), a2.cend() );
49    cout << "\nThe minimum and maximum elements in a2 are "
50       << *minAndMax.first << " and " << *minAndMax.second
51       << ", respectively";
52
53    // calculate sum of elements in a1
54    cout << "\n\nThe total of the elements in a1 is: "
55       << accumulate( a1.cbegin(), a1.cend(), 0 );
56
57    // output square of every element in a1
58    cout << "\n\nThe square of every integer in a1 is:\n";
59    for_each( a1.cbegin(), a1.cend(), outputSquare );
60
61    array< int, SIZE > cubes; // instantiate cubes
62
63    // calculate cube of each element in a1; place results in cubes
64    transform( a1.cbegin(), a1.cend(), cubes.begin(), calculateCube );
65    cout << "\n\nThe cube of every integer in a1 is:\n";
66    copy( cubes.cbegin(), cubes.cend(), output );
67    cout << endl;
68 } // end main
69
```

Fig. 16.5 | Mathematical algorithms of the Standard Library. (Part 3 of 5.)


```
70 // determine whether argument is greater than 9
71 bool greater9( int value )
72 {
73    return value > 9;
74 } // end function greater9
75
76 // output square of argument
77 void outputSquare( int value )
78 {
79    cout << value * value << ' ';
80 } // end function outputSquare
81
82 // return cube of argument
83 int calculateCube( int value )
84 {
85    return value * value * value;
86 } // end function calculateCube
```

Fig. 16.5 | Mathematical algorithms of the Standard Library. (Part 4 of 5.)


```
a1 before random_shuffle: 1 2 3 4 5 6 7 8 9 10
a1 after random_shuffle: 9 2 10 3 1 6 8 4 5 7

a2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in a2 is: 1
Maximum element in a2 is: 100
The minimum and maximum elements in a2 are 1 and 100, respectively

The total of the elements in a1 is: 55

The square of every integer in a1 is:
81 4 100 9 1 36 64 16 25 49

The cube of every integer in a1 is:
729 8 1000 27 1 216 512 64 125 343
```

Fig. 16.5 | Mathematical algorithms of the Standard Library. (Part 5 of 5.)

16.3.5 Mathematical Algorithms (Cont.)

`accumulate` Algorithm

- Line 55 uses the accumulate algorithm (the template of which is in header `<numeric>`) to sum the values in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`.
- The algorithm's two iterator arguments must be at least *input iterators* and its third argument represents the initial value of the total.
- A second version of this algorithm takes as its fourth argument a general function that determines how elements are accumulated.
- The general function must take *two* arguments and return a result.

16.3.5 Mathematical Algorithms (Cont.)

***for_each* Algorithm**

- Line 59 uses the for_each algorithm to apply a general function to every element in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`.
- The general function takes the current element as an argument and may modify that element (if it's received by reference and is not `const`).
- Algorithm `for_eah` requires its two iterator arguments to be at least *input iterators*.

Use accumulate or for_each to summarize ranges

16.3.6 Basic Searching and Sorting Algorithms

- Figure 16.6 demonstrates some basic searching and sorting capabilities of the Standard Library, including `find`, `find_if`, `sort`, `binary_search`, `all_of`, `any_of`, `none_of` and `find_if_not`.


```cpp
// Fig. 16.6: fig16_06.cpp
// Standard Library search and sort algorithms.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <array> // array class-template definition
#include <iterator>
using namespace std;

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

int main()
{
   const size_t SIZE = 10;
   array< int, SIZE > a = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
   ostream_iterator< int > output( cout, " " );

   cout << "array a contains: ";
   copy( a.cbegin(), a.cend(), output ); // display output vector

   // locate first occurrence of 16 in a
   auto location = find( a.cbegin(), a.cend(), 16 );

```

Fig. 16.6 | Standard Library search and sort algorithms. (Part 1 of 6.)


```cpp
   if ( location != a.cend() ) // found 16
      cout << "\n\nFound 16 at location " << ( location - a.cbegin() );
   else // 16 not found
      cout << "\n\n16 not found";

   // locate first occurrence of 100 in a
   location = find( a.cbegin(), a.cend(), 100 );

   if ( location != a.cend() ) // found 100
      cout << "\nFound 100 at location " << ( location - a.cbegin() );
   else // 100 not found
      cout << "\n100 not found";

   // locate first occurrence of value greater than 10 in a
   location = find_if( a.cbegin(), a.cend(), greater10 );

   if ( location != a.cend() ) // found value greater than 10
      cout << "\n\nThe first value greater than 10 is " << *location
         << "\nfound at location " << ( location - a.cbegin() );
   else // value greater than 10 not found
      cout << "\n\nNo values greater than 10 were found";

```

Fig. 16.6 | Standard Library search and sort algorithms. (Part 2 of 6.)

```
45      // sort elements of a
46      sort( a.begin(), a.end() );
47      cout << "\n\narray a after sort: ";
48      copy( a.cbegin(), a.cend(), output );
49
50      // use binary_search to locate 13 in a
51      if ( binary_search( a.cbegin(), a.cend(), 13 ) )
52         cout << "\n\n13 was found in a";
53      else
54         cout << "\n\n13 was not found in a";
55
56      // use binary_search to locate 100 in a
57      if ( binary_search( a.cbegin(), a.cend(), 100 ) )
58         cout << "\n100 was found in a";
59      else
60         cout << "\n100 was not found in a";
61
62      // determine whether all of the elements of a are greater than 10
63      if ( all_of( a.cbegin(), a.cend(), greater10 ) )
64         cout << "\n\nAll the elements in a are greater than 10";
65      else
66         cout << "\n\nSome elements in a are not greater than 10";
67
```

Fig. 16.6 | Standard Library search and sort algorithms. (Part 3 of 6.)


```
68      // determine whether any of the elements of a are greater than 10
69      if ( any_of( a.cbegin(), a.cend(), greater10 ) )
70         cout << "\n\nSome of the elements in a are greater than 10";
71      else
72         cout << "\n\nNone of the elements in a are greater than 10";
73
74      // determine whether none of the elements of a are greater than 10
75      if ( none_of( a.cbegin(), a.cend(), greater10 ) )
76         cout << "\n\nNone of the elements in a are greater than 10";
77      else
78         cout << "\n\nSome of the elements in a are greater than 10";
79
80      // locate first occurrence of value that's not greater than 10 in a
81      location = find_if_not( a.cbegin(), a.cend(), greater10 );
82
83      if ( location != a.cend() ) // found a value less than or eqaul to 10
84         cout << "\n\nThe first value not greater than 10 is " << *location
85            << "\nfound at location " << ( location - a.cbegin() );
86      else // no values less than or equal to 10 were found
87         cout << "\n\nOnly values greater than 10 were found";
88
89      cout << endl;
90   } // end main
91
```

Fig. 16.6 | Standard Library search and sort algorithms. (Part 4 of 6.)


```
92   // determine whether argument is greater than 10
93   bool greater10( int value )
94   {
95      return value > 10;
96   } // end function greater10
```

Fig. 16.6 | Standard Library search and sort algorithms. (Part 5 of 6.)

16.3.6 Basic Searching and Sorting Algorithms (Cont.)

`find` Algorithm

- Line 21 uses the find algorithm to locate the value `16` in the range from `a.cbegin()` up to, but not including, `a.cend()`.
- The algorithm requires its two iterator arguments to be at least *input iterators* and returns an *input iterator* that either is positioned at the first element containing the value or indicates the end of the sequence (as is the case in line 29).

16.3.6 Basic Searching and Sorting Algorithms (Cont.)

find_if Algorithm

- Line 37 uses the find_if algorithm (a linear search)to locate the first value in the range from `a.cbegin()` up to, but *not* including, `a.cend()` for which the *unary predicate function* `greater10` returns `true`.
- Function `greater10` (defined in lines 93–96) takes an integer and returns a `bool` value indicating whether the integer argument is *greater than* 10.
- Algorithm `find_if` requires its two iterator arguments to be at least *input iterators*.
- The algorithm returns an *input iterator* that either is positioned at the first element containing a value for which the predicate function returns `true` or indicates the end of the sequence.

16.3.6 Basic Searching and Sorting Algorithms (Cont.)

sort Algorithm

- Line 46 uses the sort algorithm to arrange the elements in the range from `a.begin()` up to, but *not* including, `a.end()` in *ascending order*.
- The algorithm requires its two iterator arguments to be *random-access iterators*.
- A second version of this algorithm takes a third argument that is a *binary predicate function* taking two arguments that are values in the sequence and returning a `bool` indicating the *sorting order*—if the return value is `true`, the two elements being compared are in *sorted order*.

16.3.6 Basic Searching and Sorting Algorithms (Cont.)

binary_search Algorithm

- Line 53 uses the binary_search algorithm to determine whether the value 13 is in the range from `a.cbegin()` up to, but *not* including, `a.cend()`.
- The values must be sorted in *ascending* order.
- Algorithm `binary_search` requires its two iterator arguments to be at least *forward iterators*.
- The algorithm returns a `bool` indicating whether the value was found in the sequence.

16.3.6 Basic Searching and Sorting Algorithms (Cont.)

- Line 57 demonstrates a call to `binary_search` in which the value is *not* found.
- A second version of this algorithm takes a fourth argument that is a *binary predicate function* taking two arguments that are values in the sequence and returning a `bool`.
- The predicate function returns `true` if the two elements being compared are in *sorted order*.
- To obtain the *location* of the search key in the container, use the `lower_bound` or `find` algorithms.

Prefer member functions to algorithms with the same names

```
set<int> s; // create set, put
         … //1,000,000 values into it

// use find member function
set<int>::iterator i = s.find(727);
if (i != s.end())...

// use find algorithm
set<int>::iterator i = find(s.begin(), s.end(), 727);
if(i!=s.end())...
```

- find member function runs in logarithmic time →About 40 (worst case) to about 20 (average case) comparisons
- find algorithm runs in linear time → 1 ,000,000 (worst case) to 500,000 (average case) comparisons

16.3.8 copy_backward, merge, unique and reverse

- Figure 16.8 demonstrates algorithms `copy_backward`, `merge`, `unique` and `reverse`.


```
// Fig. 16.8: fig16_08.cpp
// Algorithms copy_backward, merge, unique and reverse.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <array> // array class-template definition
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const size_t SIZE = 5;
   array< int, SIZE > a1 = { 1, 3, 5, 7, 9 };
   array< int, SIZE > a2 = { 2, 4, 5, 7, 9 };
   ostream_iterator< int > output( cout, " " );

   cout << "array a1 contains: ";
   copy( a1.cbegin(), a1.cend(), output ); // display a1
   cout << "\narray a2 contains: ";
   copy( a2.cbegin(), a2.cend(), output ); // display a2

   array< int, SIZE > results;

```

Fig. 16.8 | Algorithms `copy_backward`, `merge`, `unique` and `reverse`. (Part 1 of 3.)


```
   // place elements of a1 into results in reverse order
   copy_backward( a1.cbegin(), a1.cend(), results.end() );
   cout << "\n\nAfter copy_backward, results contains: ";
   copy( results.cbegin(), results.cend(), output );

   array< int, SIZE + SIZE >  results2;

   // merge elements of a1 and a2 into results2 in sorted order
   merge( a1.cbegin(), a1.cend(), a2.cbegin(), a2.cend(),
      results2.begin() );

   cout << "\n\nAfter merge of a1 and a2 results2 contains: ";
   copy( results2.cbegin(), results2.cend(), output );

   // eliminate duplicate values from results2
   auto endLocation = unique( results2.begin(), results2.end() );

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

```

Fig. 16.8 | Algorithms `copy_backward`, `merge`, `unique` and `reverse`. (Part 2 of 3.)

```
43     cout << "\n\narray a1 after reverse: ";
44     reverse( a1.begin(), a1.end() ); // reverse elements of a1
45     copy( a1.cbegin(), a1.cend(), output );
46     cout << endl;
47  } // end main
```

```
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9

After copy_backward, results contains: 1 3 5 7 9

After merge of a1 and a2 results2 contains: 1 2 3 4 5 5 7 7 9 9

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

array a1 after reverse: 9 7 5 3 1
```

Fig. 16.8 | Algorithms `copy_backward`, `merge`, `unique` and `reverse`. (Part 3 of 3.)

16.3.8 copy_backward, merge, unique and reverse

copy_backward Algorithm

- Line 24 uses the copy_backward algorithm to copy elements in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`, placing the elements in `results` by starting from the element before `results.end()` and working toward the beginning of the `array`.
- The algorithm returns an iterator positioned at the *last* element copied into the `results` (i.e., the beginning of `results`, because of the backward copy).
- The elements are placed in `results` in the same order as `a1`.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

- This algorithm requires three *bidirectional iterator* arguments (iterators that can be *incremented* and *decremented* to iterate *forward* and *backward* through a sequence, respectively).
- One difference between `copy_backward` and `copy` is that the iterator returned from `copy` is positioned after the last element copied and the one returned from `copy_backward` is positioned *at* the last element copied (i.e., the first element in the sequence).
- Also, `copy_backward` *can* manipulate *overlapping* ranges of elements in a container as long as the first element to copy is *not* in the destination range of elements.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

- In addition to the `copy` and `copy_backward` algorithms, C++11 now includes the move and `move_backward` algorithms.
- These use C++11's new move semantics (discussed in Chapter 24, C++11: Additional Features) to move, rather than copy, objects from one container to another.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

merge ***Algorithm***

- Lines 31-32 use the merge algorithm to combine two *sorted ascending sequences* of values into a third sorted ascending sequence.
- The algorithm requires five iterator arguments.
- The first four must be at least *input iterators* and the last must be at least an *output iterator*.
- The first two arguments specify the range of elements in the first sorted sequence (`a1`), the second two arguments specify the range of elements in the second sorted sequence (`a2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged.
- A second version of this algorithm takes as its sixth argument a *binary predicate function* that specifies the *sorting order*.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

***back_inserter*, *front_inserter* and *inserter* Iterator Adapters**

- Line 28 creates the `array results2` with the number of elements in `a1` and `a2`.
- Using the `merge` algorithm requires that the sequence where the results are stored be at least the size of the sequences being merged.
- If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation, you can use the following statements:

```
vector< int > results2;
merge( a1.begin(), a1.end(), a2.begin(), a2.end(),
   back_inserter( results2 ) );
```


16.3.8 copy_backward, merge, unique and reverse (Cont.)

- The argument `back_inserter(results2)` uses function template back_in-serter (header `<iterator>`) for the `vector results2`.
- A `back_in-serter` calls the container's default `push_back` function to insert an element at the end of the container.
- If an element is inserted into a container that has no more space available, *the container grows in size*—which is why we used a `vector` in the preceding statements, because `array`s are fixed size.
- Thus, the number of elements in the container does *not* have to be known in advance.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

- There are two other inserters—front_inserter (uses `push_front` to insert an element at the *beginning* of a container specified as its argument) and inserter (uses `insert` to insert an element *at* the iterator supplied as its second argument in the container supplied as its first argument).

16.3.8 copy_backward, merge, unique and reverse (Cont.)

***unique* Algorithm**

- Line 38 uses the unique algorithm on the *sorted* sequence of elements in the range from `results2.begin()` up to, but *not* including, `results2.end()`.
- After this algorithm is applied to a sorted sequence with *duplicate* values, only a *single* copy of each value remains in the sequence.
- The algorithm takes two arguments that must be at least *forward iterators*.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

- The algorithm returns an iterator positioned *after the last element* in the sequence of unique values.
- The values of all elements in the container after the last unique value are *undefined*.
- A second version of this algorithm takes as a third argument a *binary predicate function* specifying how to compare two elements for *equality*.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

***reverse* Algorithm**

- Line 44 uses the reverse algorithm to reverse all the elements in the range from `a1.begin()` up to, but *not* including, `a1.end()`.
- The algorithm takes two arguments that must be at least *bidirectional iterators*.

16.3.8 copy_backward, merge, unique and reverse (Cont.)

C++11: copy_if and copy_n Algorithms

- The `copy_if` algorithm copies each element from a range if the *unary predicate function* in its fourth argument returns `true` for that element.
- The iterators supplied as the first two arguments must be *input iterators*.
- The iterator supplied as the third argument must be an *output iterator* so that the element being copied can be assigned to the copy location.
- This algorithm returns an iterator positioned after the *last* element copied.
- The `copy_n` algorithm copies the number of elements specified by its second argument from the location specified by its first argument (an *input iterator*).
- The elements are output to the location specified by its third argument (an *output iterator*).

16.4 Function Objects

- Many Standard Library algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task.
- For example, the `binary_search` algorithm that we discussed in Section 16.3.6 is overloaded with a version that requires as its fourth parameter a *function pointer* that takes two arguments and returns a `bool` value.
- The algorithm uses this function to compare the search key to an element in the collection.
- The function returns `true` if the search key and element being compared are equal; otherwise, the function returns `false`.

16.4 Function Objects (Cont.)

- This enables `binary_search` to search a collection of elements for which the element type does *not* provide an overloaded equality < operator.
- Any algorithm that can receive a *function pointer* can also receive an object of a class that overloads the function-call operator (parentheses) with a function named `operator()`, provided that the overloaded operator meets the requirements of the algorithm—in the case of `binary_search`, it must receive two arguments and return a `bool`.

16.4 Function Objects (Cont.)

- An object of such a class is known as a function object and can be used syntactically and semantically like a function or *function pointer*—the overloaded parentheses operator is invoked by using a function object's name followed by parentheses containing the arguments to the function.
- Most algorithms can use function objects and functions interchangeably.

16.4 Function Objects (Cont.)

Real life example

```
typedef std::shared_ptr<DomProject> ProjectPtr;
typedef std::unordered_map<unsigned short,ProjectPtr> ProjectLijst;
std::vector<ProjectPtr> lijstProjecten; // member of ProjectListWidget
struct sorteerPL
{
  bool operator() (ProjectPtr p1,ProjectPtr p2)
  {
      if (p1->KlantCode() != p2->KlantCode())
         return p1->KlantCode() < p2->KlantCode();
      else
         return p1->Omschrijving() < p2->Omschrijving();
  }
} sorteerPLObj;

void ProjectListWidget::SetLijstProjecten()
{                                              // convert unordered_map
    for (auto p:DomProject::projectLijst)  // to sorted vector
    {
        lijstProjecten.push_back(p.second);
    }
    sort(lijstProjecten.begin(),lijstProjecten.end(),sorteerPLObj);
}
```

16.4 Function Objects (Cont.)

Advantages of Function Objects Over Function Pointers

- *Function objects* provide several advantages over *function pointers*.
- The compiler can inline a *function object's* overloaded `operator()` to improve performance.
- Also, since they're objects of classes, *function objects* can have data members that `operator()` can use to perform its task.

16.4 Function Objects (Cont.)

Predefined Function Objects of the Standard Template Library

- Many predefined function objects can be found in the header `<functional>`.
- Figure 16.14 lists several of the dozens of Standard Library *function objects*, which are all implemented as class templates.
- We used the *function object* `less< T >` in the `set`, `multiset` and `priority_queue` examples, to specify the sorting order for elements in a container.

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

Fig. 16.14 | Function objects in the Standard Library.

16.4 Function Objects (Cont.)

Using the `accumulate` Algorithm

- Figure 16.15 uses the `accumulate` numeric algorithm (introduced in Fig. 16.30) to calculate the sum of the squares of the elements in an `array`.
- The fourth argument to `accumulate` is a binary function object (that is, a *function object* for which `operator()` takes two arguments) or a function pointer to a binary function (that is, a function that takes two arguments).
- Function `accumulate` is demonstrated twice—once with a *function pointer* and once with a *function object*.

```cpp
// Fig. 16.15: fig16_15.cpp
// Demonstrating function objects.
#include <iostream>
#include <array> // array class-template definition
#include <algorithm> // copy algorithm
#include <numeric> // accumulate algorithm
#include <functional> // binary_function definition
#include <iterator> // ostream_iterator
using namespace std;

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

```

Fig. 16.15 | Binary function object. (Part 1 of 4.)


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

int main()
{
   const size_t SIZE = 10;
   array< int, SIZE > integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   ostream_iterator< int > output( cout, " " );

   cout << "array integers contains:\n";
   copy( integers.cbegin(), integers.cend(), output );

```

Fig. 16.15 | Binary function object. (Part 2 of 4.)


```cpp
   // calculate sum of squares of elements of array integers
   // using binary function sumSquares
   int result = accumulate( integers.cbegin(), integers.cend(),
      0, sumSquares );

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

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

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

Fig. 16.15 | Binary function object. (Part 3 of 4.)


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

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

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

Fig. 16.15 | Binary function object. (Part 4 of 4.)

16.4 Function Objects (Cont.)

Function ***`sumSquares`***

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

16.4 Function Objects (Cont.)

Class ***`SumSquaresClass`***

- Lines 21–30 define `SumSquaresClass` with an overloaded `operator()` that has two parameters and returns a value—the requirements for a binary function object.
- On the first call to the *function object*, the first argument will be the initial value of the `total` (which is supplied as the third argument to accumulate; `0` in this program) and the second argument will be the first element in `array integers`.
- All subsequent calls to `operator()` receive as the first argument the result returned by the previous call to the *function object*, and the second argument will be the next element in the `array`.
- When `accumulate` completes, it returns the sum of the squares of all the elements in the `array`.

16.4 Function Objects (Cont.)

Passing Function Pointers and Function Objects to Algorithm ***`accumulate`***

- Lines 43–44 call function `accumulate` with a *pointer to function* `sumSquares` as its last argument.
- Similarly, the statement in lines 51–52 calls `accumulate` with an object of class `SumSquaresClass` as the last argument.
- The expression `SumSquaresClass< int >()` creates (and calls the default constructor for) an instance of class `SumSquaresClass` (a *function object*) that is passed to `accumulate`, which invokes function `operator()`.

16.4 Function Objects: ex. with datamember

```
class StringAppender {
public:
/* Constructor takes and stores a string. */
explicit StringAppender(const string& str) : toAppend(str) {}
/* operator() prints out a string, plus the stored suffix. */
void operator() (const string& str) const {
cout << str << ' ' << toAppend << endl;
}
private:
const string toAppend;
};

StringAppender myFunctor("is awesome");
myFunctor("C++");   // prints out: "C++ is awesome"
```

16.5 Lambda Expressions

- Before you can pass a function pointer or function object to an algorithm, the corresponding function or class must have been declared.
- C++11's Lambda expressions (or lambda functions) enable you to define anonymous function objects where they're passed to a function.
- They're defined locally inside functions and can "capture" (by value or by reference) the local variables of the enclosing function then manipulate these variables in the lambda's body.
- Figure 16.16 demonstrates a simple lambda expression example that doubles the value of each element in an `int` array.


```cpp
1  // Fig. 16.16: fig16_16.cpp
2  // Lambda expressions.
3  #include <iostream>
4  #include <array>
5  #include <algorithm>
6  using namespace std;
7
8  int main()
9  {
10    const size_t SIZE = 4; // size of array values
11    array< int, SIZE >  values = { 1, 2, 3, 4 }; // initialize values
12
13    // output each element multiplied by two
14    for_each( values.cbegin(), values.cend(),
15       []( int i ) { cout << i * 2 << endl; } );
16
17    int sum = 0; // initialize sum to zero
18
19    // add each element to sum
20    for_each( values.cbegin(), values.cend(),
21       [ &sum ]( int i ) { sum += i; } );
22
23    cout << "sum is " << sum << endl; // output sum
24 } // end main
```

Fig. 16.16 | Lambda expressions. (Part 1 of 2.)


```
2
4
6
8
sum is 10
```

Fig. 16.16 | Lambda expressions. (Part 2 of 2.)

16.5 Lambda Expressions (cont.)

- Lines 10 and 11 declare and initialize a small array of `int`s named `values`.
- Lines 14–15 call the `for_each` algorithm on the elements of values.
- The third argument (line 15) to `for_each` is a *lambda expression*.
- Lambdas begin with *lambda introducer* (`[]`), followed by a parameter list and function body.
- Return types can be inferred automatically if the body is a single statement of the form `return` *expression*;—otherwise, the return type is `void` by default or you can explicitly use a *trailing return type*.
- The compiler converts the lambda expression into a function object. The lambda expression in line 15 receives an `int`, multiplies it by 2 and displays the result.
- The `for_each` algorithm passes each element of the array to the lambda.

16.5 Lambda Expressions (cont.)

- The second call to the `for_each` algorithm (lines 20–21) calculates the sum of the `array` elements.
- The lambda introducer `[&sum]` indicates that this lambda expression *captures* the local variable `sum` *by reference* (note the use of the ampersand), so that the lambda can modify `sum`'s value.
- Without the ampersand, `sum` would be captured by `value` and the local variable outside the lambda expression would *not* be updated.
- The `for_each` algorithm passes each element of values to the lambda, which adds the value to the `sum`. Line 23 then displays the value of `sum`.

16.5 Lambda Expressions (cont.)

- You can assign lambda expressions to variables, which can then be used to invoke the lambda expression or pass it to other functions.
- For example, you can assign the lambda expression in line 15 to a variable as follows:

```
auto myLambda = []( int i ) { cout << i * 2 << endl; };
```

- You can then use the variable name as a function name to invoke the lambda as in:

```
myLambda( 10 ); // outputs 20
```


16.5 Lambda Expressions (cont.)

Real life example

```
void DomFactuurWriteTSTekst::Do(const TimesheetLijst &tsl)
{
// …
tsFile << "Bijlage bij factuur " << factuur->Nr() << " van "
       << factuur->FactuurDatum().toString(Qt::SystemLocaleShortDate).toStdString()
       << endl;
 auto OutputTS = [this,&totaalDuur,&totaalKm] (TimesheetPtr ts)
 {
  tsFile << setw(12) << left
               << ts->Datum().toString(Qt::SystemLocaleShortDate).toStdString()
               << setw(35) << left << ts->Oms()
               << setw(5)  << right
               << ts->Van().toString(Qt::SystemLocaleShortDate).toStdString()
               << setw(7)  << right
               << ts->Tot().toString(Qt::SystemLocaleShortDate).toStdString()
               << setw(10) << right << setprecision(2) << fixed << ts->Duur()
               << setw(10)  << right << factuur->Project()->Klant()->Uurtarief()
               << endl;
  totaalDuur += ts->Duur();
  totaalKm += ts->Km();
 };
 for_each (tsl.cbegin(),tsl.cend(),OutputTS);
}
```


16.5 Lambda Expressions (cont.)

3 methods to copy unordered_map to sorted vector

Real life example

```
// defined in some header files:
typedef std::shared_ptr<DomProject> ProjectPtr;
typedef std::unordered_map<unsigned short,ProjectPtr> ProjectLijst;
static ProjectLijst projectLijst;  // this list has been filled
// local function: 3 methods to copy unordered_map to sorted vector
vector<ProjectPtr> pLijstSorted;

/* C++11 range-based for loop and auto type deduction */
for (auto p:DomProject::projectLijst) {
       pLijstSorted.push_back(p.second);
}
/* C++11  lambda expression */
for_each(DomProject::projectLijst.cbegin(),DomProject::projectLijst.cend(),
        [&pLijstSorted] (const pair<unsigned short,ProjectPtr>& p)
        {pLijstSorted.push_back(p.second);});
// C++ 14 generic lambda expression
for_each(DomProject::projectLijst.cbegin(),DomProject::projectLijst.cend(),
                         [&pLijstSorted] (const auto& p)
                         {pLijstSorted.push_back(p.second);});

 sort(pLijstSorted.begin(),pLijstSorted.end(),DomProject::sorteerPLObj);
               // sorteerPLObj is a predefined function (see §16.4)
```

16.6 Standard Library Algorithm Summary

- The C++ standard specifies over 90 algorithms—many overloaded with two or more versions.
- The standard separates the algorithms into several categories
  - mutating sequence algorithms
  - nonmodifying sequence algorithms
  - sorting and related algorithms
  - generalized numeric operations.
- To learn about the algorithms that we did not present in this chapter, see your compiler's documentation or visit sites such as
  - `en.cppreference.com/w/cpp/algorithm`

16.6 Standard Library Algorithm Summary (cont.)

Mutating Sequence Algorithms

- Figure 16.17 shows many of the mutating-sequence algorithms—i.e., algorithms that modify the containers they operate on.
- Algorithms new in C++11 are marked with an * in Figs. 16.17–16.20.
- Algorithms presented in this chapter are shown in **`bold`**.

Mutating sequence algorithms from header <algorithm>			
copy	copy_n*	copy_if*	**copy_backward**
move*	move_backward*	**swap**	**swap_ranges**
iter_swap	**transform**	**replace**	**replace_if**
replace_copy	**replace_copy_if**	**fill**	**fill_n**
generate	**generate_n**	**remove**	**remove_if**
remove_copy	**remove_copy_if**	**unique**	**unique_copy**
reverse	**reverse_copy**	rotate	rotate_copy
random_shuffle	shuffle*	is_partitioned*	partition
stable_partition	partition_copy*	partition_point*	

Fig. 16.17 | Mutating-sequence algorithms from header <algorithm>.

16.6 Standard Library Algorithm Summary (cont.)

Nonmodifying Sequence Algorithms

- Figure 16.18 shows the nonmodifying sequence algorithms—i.e., algorithms that do *not* modify the containers they operate on.

Nonmodifying sequence algorithms from header <algorithm>			
all_of*	any_of*	none_of*	for_each
find	find_if	find_if_not*	find_end
find_first_of	adjacent_find	count	count_if
mismatch	equal	is_permutation*	search
search_n			

Fig. 16.18 | Nonmodifying sequence algorithms from header <algorithm>.

16.6 Standard Library Algorithm Summary (cont.)

Sorting and Related Algorithms

- Figure 16.19 shows the *sorting and related algorithms*.

Sorting and related algorithms from header <algorithm>			
sort	stable_sort	partial_sort	partial_sort_copy
is_sorted*	is_sorted_until*	nth_element	lower_bound
upper_bound	equal_range	binary_search	merge
inplace_merge	includes	set_union	set_intersection
set_difference	set_symmetric_difference		push_heap
pop_heap	make_heap	sort_heap	is_heap*
is_heap_until*	min	max	minmax*
min_element	max_element	minmax_element*	lexicographical_compare
next_permutation	prev_permutation		

Fig. 16.19 | Sorting and related algorithms from header <algorithm>.

16.6 Standard Library Algorithm Summary (cont.)

Numerical Algorithms

- Figure 16.20 shows the numerical algorithms of the header `<numeric>`.

Numerical algorithms from header <numeric>		
accumulate	partial_sum	iota*
inner_product	adjacent_difference	

Fig. 16.20 | Numerical algorithms from header <numeric>.