Lecture 32:

Standard Template Library (STL)

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 21st, 2010

22.2.1 vector Sequence Container

- Figure 22.15 illustrates functions that enable retrieval and manipulation of the elements of a `vector`.
- Line 15 uses an overloaded `vector` constructor that takes two iterators as arguments to initialize `integers`.

```cpp
// Fig. 22.15: Fig22_15.cpp
// Testing Standard Library vector class template
// element-manipulation functions.
#include <iostream>
#include <vector> // vector class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator iterator
#include <stdexcept> // out_of_range exception
using namespace std;

int main()
{
   const int SIZE = 6;
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > integers( array, array + SIZE );
   ostream_iterator< int > output( cout, " " );

```

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 1 of 4.)

```
   cout << "Vector integers contains: ";
   copy( integers.begin(), integers.end(), output );

   cout << "\nFirst element of integers: " << integers.front()
      << "\nLast element of integers: " << integers.back();

   integers[ 0 ] = 7; // set first element to 7
   integers.at( 2 ) = 10; // set element at position 2 to 10

   // insert 22 as 2nd element
   integers.insert( integers.begin() + 1, 22 );

   cout << "\n\nContents of vector integers after changes: ";
   copy( integers.begin(), integers.end(), output );

   // access out-of-range element
   try
   {
      integers.at( 100 ) = 777;
   } // end try
   catch ( out_of_range &outOfRange ) // out_of_range exception
   {
      cout << "\n\nException: " << outOfRange.what();
   } // end catch
```

Fig. 22.15 | vector class template element-manipulation functions. (Part 2 of 4.)

```
    // erase first element
    integers.erase( integers.begin() );
    cout << "\n\nVector integers after erasing first element: ";
    copy( integers.begin(), integers.end(), output );

    // erase remaining elements
    integers.erase( integers.begin(), integers.end() );
    cout << "\nAfter erasing all elements, vector integers "
       << ( integers.empty() ? "is" : "is not" ) << " empty";

    // insert elements from array
    integers.insert( integers.begin(), array, array + SIZE );
    cout << "\n\nContents of vector integers before clear: ";
    copy( integers.begin(), integers.end(), output );

    // empty integers; clear calls erase to empty a collection
    integers.clear();
    cout << "\nAfter clear, vector integers "
       << ( integers.empty() ? "is" : "is not" ) << " empty" << endl;
} // end main
```

Fig. 22.15 | vector class template element-manipulation functions. (Part 3 of 4.)

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

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

Exception: invalid vector<T> subscript

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

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

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 4 of 4.)

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

Fig. 22.16 | Some STL exception types.

22.2.1 vector Sequence Container

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

22.2.1 vector Sequence Container

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

Common Programming Error 22.4

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

22.2.1 vector Sequence Container

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

22.2.1 vector Sequence Container

- Line 54 demonstrates the version of function `insert` that uses the second and third arguments to specify the starting location and ending location in a sequence of values (possibly from another container; in this case, from array of integers `array`) that should be inserted into the `vector`.
- Remember that the ending location specifies the position in the sequence after the last element to be inserted; copying is performed up to—but not including—this location.
- Finally, line 59 uses function `clear` (found in all first-class containers) to empty the `vector`.
- This function calls the version of `erase` used in line 51 to empty the `vector`.

22.2.2 `list` Sequence Container

- The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container.
- If most of the insertions and deletions occur at the ends of the container, the `deque` data structure (Section 22.2.3) provides a more efficient implementation.
- Class template `list` is implemented as a doubly linked list—every node in the `list` contains a pointer to the previous node in the `list` and to the next node in the `list`.
- This enables class template `list` to support bidirectional iterators that allow the container to be traversed both forward and backward.

22.2.2 `list` Sequence Container (Cont.)

- Any algorithm that requires input, output, forward or bidirectional iterators can operate on a `list`.
- Many `list` member functions manipulate the elements of the container as an ordered set of elements.
- In addition to the member functions of all STL containers in Fig. 22.2 and the common member functions of all sequence containers discussed in Section 22.2, class template `list` provides nine other member functions—`splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`.

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

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

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

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

```

Fig. 22.17 | Standard Library `list` class template. (Part 1 of 6.)

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

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

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

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

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

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   otherValues.sort();
```

Fig. 22.17 | Standard Library `list` class template. (Part 2 of 6.)

```
   cout << "\nAfter insert and sort, otherValues contains: ";
   printList( otherValues );

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

   values.pop_front(); // remove element from front
   values.pop_back(); // remove element from back
   cout << "\nAfter pop_front and pop_back:\n   values contains: "
   printList( values );

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

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

Fig. 22.17 | Standard Library `list` class template. (Part 3 of 6.)

```
    printList( otherValues );

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

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

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

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

Fig. 22.17 | Standard Library `list` class template. (Part 4 of 6.)

```
    else
    {
       ostream_iterator< T > output( cout, " " );
       copy( listRef.begin(), listRef.end(), output );
    } // end else
} // end function printList
```

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

Fig. 22.17 | Standard Library `list` class template. (Part 5 of 6.)

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

Fig. 22.17 | Standard Library `list` class template. (Part 6 of 6.)

22.2.2 `list` Sequence Container (Cont.)

- Several of these member functions are `list`-optimized implementations of STL algorithms presented in Section 22.5.
- Figure 22.17 demonstrates several features of class `list`.
- Remember that many of the functions presented in Figs. 22.14–22.15 can be used with class `list`.
- Header file `<list>` must be included to use class `list`.

22.2.2 list Sequence Container (Cont.)

- Lines 16–17 instantiate two list objects capable of storing integers.
- Lines 20–21 use function push_front to insert integers at the beginning of values.
- Function push_front is specific to classes list and deque (not to vector).
- Lines 22–23 use function push_back to insert integers at the end of values.
- Remember that function push_back is common to all sequence containers.
- Line 28 uses list member function sort to arrange the elements in the list in ascending order.

22.2.2 `list` Sequence Container (Cont.)

- A second version of function `sort` allows you to supply a binary predicate function that takes two arguments (values in the list), performs a comparison and returns a `bool` value indicating the result.
- This function determines the order in which the elements of the `list` are sorted.
- This version could be particularly useful for a `list` that stores pointers rather than values.
- [*Note: We demonstrate a unary predicate*

22.2.2 `list` Sequence Container (Cont.)

- A unary predicate function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.]
- Line 38 uses `list` function `splice` to remove the elements in `otherValues` and insert them into `values` before the iterator position specified as the first argument.
- There are two other versions of this function.
- Function `splice` with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument.

22.2.2 `list` Sequence Container (Cont.)

- Function `splice` with four arguments uses the last two arguments to specify a range of locations that should be removed from the container in the second argument and placed at the location specified in the first argument.
- After inserting more elements in `otherValues` and sorting both `values` and `other-Values`, line 53 uses `list` member function merge to remove all elements of `otherValues` and insert them in sorted order into `values`.
- Both `list`s must be sorted in the same order before this operation is performed.
- A second version of `merge` enables you to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value.

22.2.2 `list` Sequence Container (Cont.)

- The predicate function specifies the sorting order used by `merge`.
- Line 59 uses `list` function `pop_front` to remove the first element in the `list`.
- Line 60 uses function `pop_back` (available for all sequence containers) to remove the last element in the `list`.
- Line 64 uses `list` function `unique` to remove duplicate elements in the `list`.
- The `list` should be in sorted order (so that all duplicates are side by side) before this operation is performed, to guarantee that all duplicates are eliminated.

22.2.2 `list` Sequence Container (Cont.)

- A second version of `unique` enables you to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value specifying whether two elements are equal.
- Line 69 uses function swap (available to all first-class containers) to exchange the contents of `values` with the contents of `otherValues`.
- Line 76 uses `list` function assign (available to all sequence containers) to replace the contents of `values` with the contents of `otherValues` in the range specified by the two iterator arguments.

22.2.2 `list` Sequence Container (Cont.)

- A second version of `assign` replaces the original contents with copies of the value specified in the second argument.
- The first argument of the function specifies the number of copies.
- Line 85 uses `list` function remove to delete all copies of the value `4` from the `list`.

22.2.3 deque Sequence Container

- Class `deque` provides many of the benefits of a `vector` and a `list` in one container.
- The term `deque` is short for “double-ended queue.”
- Class `deque` is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a `vector`.
- Class `deque` is also implemented for efficient insertion and deletion operations at its front and back, much like a `list` (although a `list` is also capable of efficient insertions and deletions in the middle of the `list`).
- Class `deque` provides support for random-access iterators, so `deque`s can be used with all STL algorithms.

Performance Tip 22.13

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

Performance Tip 22.14

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

22.2.3 deque Sequence Container (Cont.)

- One of the most common uses of a `deque` is to maintain a first-in, first-out queue of elements.
- In fact, a `deque` is the default underlying implementation for the `queue` adaptor (Section 22.4.2).
- Additional storage for a `deque` can be allocated at either end of the `deque` in blocks of memory that are typically maintained as an array of pointers to those blocks.
- Due to the noncontiguous memory layout of a `deque`, a `deque` iterator must be more intelligent than the pointers that are used to iterate through `vector`s or pointer-based arrays.

```cpp
// Fig. 22.18: Fig22_18.cpp
// Standard Library class deque test program.
#include <iostream>
#include <deque> // deque class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   deque< double > values; // create deque of doubles
   ostream_iterator< double > output( cout, " " );

   // insert elements in values
   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );

   cout << "values contains: ";

```

Fig. 22.18 | Standard Library **deque** class template. (Part 1 of 2.)

```
   // use subscript operator to obtain elements of values
   for ( unsigned int i = 0; i < values.size(); i++ )
      cout << values[ i ] << ' ';

   values.pop_front(); // remove first element
   cout << "\nAfter pop_front, values contains: ";
   copy( values.begin(), values.end(), output );

   // use subscript operator to modify element at location 1
   values[ 1 ] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
   copy( values.begin(), values.end(), output );
   cout << endl;
} // end main
```

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

Fig. 22.18 | Standard Library **deque** class template. (Part 2 of 2.)

22.2.3 deque Sequence Container (Cont.)

- Class `deque` provides the same basic operations as class `vector`, but like `list` adds member functions `push_front` and `pop_front` to allow insertion and deletion at the beginning of the `deque`, respectively.
- Figure 22.18 demonstrates features of class `deque`.
- Remember that many of the functions presented in Fig. 22.14, Fig. 22.15 and Fig. 22.17 also can be used with class `deque`.
- Header file <deque> must be included to use class `deque`.
- Line 11 instantiates a `deque` that can store `double` values.
- Lines 15–17 use functions `push_front` and `push_back` to insert elements at the beginning and end of the `deque`.

22.2.3 deque Sequence Container (Cont.)

- The `for` statement in lines 22–23 uses the subscript operator to retrieve the value in each element of the `deque` for output.
- The condition uses function `size` to ensure that we do not attempt to access an element outside the bounds of the `deque`.
- Line 25 uses function `pop_front` to demonstrate removing the first element of the `deque`.
- Remember that `pop_front` is available only for class `list` and class `deque` (not for class `vector`).
- Line 30 uses the subscript operator to create an *lvalue*.
- This enables values to be assigned directly to any element of the `deque`.

22.3 Associative Containers

- The STL's associative containers provide direct access to store and retrieve elements via keys (often called search keys).
- The four associative containers are `multiset`, `set`, `multimap` and `map`.
- Each associative container maintains its keys in sorted order.
- Iterating through an associative container traverses it in the sort order for that container.
- Classes `multiset` and `set` provide operations for manipulating sets of values where the values are the keys—there is not a separate value associated with each key.
- The primary difference between a `multiset` and a `set` is that a `multiset` allows duplicate keys and a `set` does not.

22.3 Associative Containers (Cont.)

- Classes `multimap` and `map` provide operations for manipulating values associated with keys (these values are sometimes referred to as mapped values).
- The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.
- In addition to the common member functions of all containers presented in Fig. 22.2, all associative containers also support several other member functions, including `find`, `lower_bound`, `upper_bound` and `count`.
- Examples of each of the associative containers and the common associative container member functions are presented in the next several subsections.

22.3.1 multiset Associative Container (Cont.)

- The `multiset` associative container provides fast storage and retrieval of keys and allows duplicate keys.
- The ordering of the elements is determined by a comparator function object.
- For example, in an integer `multiset`, elements can be sorted in ascending order by ordering the keys with comparator function object `less<int>`.
- We discuss function objects in detail in Section 22.7.
- The data type of the keys in all associative containers must support comparison properly based on the comparator function object specified—keys sorted with `less< T >` must support comparison with `operator<`.

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

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

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

```

Fig. 22.19 | Standard Library `multiset` class template. (Part 1 of 4.)

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

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

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

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

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

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

```

Fig. 22.19 | Standard Library `multiset` class template. (Part 2 of 4.)

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

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

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

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

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

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

Fig. 22.19 | Standard Library `multiset` class template. (Part 3 of 4.)

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

Found value 15
Did not find value 20

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

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

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

Fig. 22.19 | Standard Library `multiset` class template. (Part 4 of 4.)

22.3.1 `multiset` Associative Container (Cont.)

- If the keys used in the associative containers are of user-defined data types, those types must supply the appropriate comparison operators.
- A `multiset` supports bidirectional iterators (but not random-access iterators).
- Figure 22.19 demonstrates the `multiset` associative container for a `multiset` of integers sorted in ascending order.
- Header file <set> must be included to use class `multiset`.
- `Containers multiset and set provide the same basic functionality.`

Good Programming Practice 22.1

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

22.3.1 `multiset` Associative Container (Cont.)

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

22.3.1 `multiset` Associative Container (Cont.)

- The output statement in line 19 uses function count (available to all associative containers) to count the number of occurrences of the value `15` currently in the `multiset`.
- Lines 22–23 use one of the three versions of function `insert` to add the value `15` to the `multiset` twice.
- A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified.
- A third version of `insert` takes two iterators as arguments that specify a range of values to add to the `multiset` from another container.

22.3.1 multiset Associative Container (Cont.)

- Line 31 uses function find (available to all associative containers) to locate the value 15 in the multiset.
- Function find returns an iterator or a const_iterator pointing to the earliest location at which the value is found.
- If the value is not found, find returns an iterator or a const_iterator equal to the value returned by a call to end.
- Line 40 demonstrates this case.
- Line 43 uses function insert to insert the elements of array a into the multiset.
- In line 45, the copy algorithm copies the elements of the multiset to the standard output in ascending order.

22.3.1 multiset Associative Container (Cont.)

- Lines 49 and 50 use functions `lower_bound` and `upper_bound` (available in all associative containers) to locate the earliest occurrence of the value `22` in the `multiset` and the element *after* the last occurrence of the value `22` in the `multiset`.
- Both functions return `iterator`s or `const_iterator`s pointing to the appropriate location or the iterator returned by `end` if the value is not in the `multiset`.
- Line 53 instantiates an instance of class `pair` called `p`.
- Objects of class `pair` are used to associate pairs of values.
- In this example, the contents of a `pair` are two `const_iterator`s for our integer-based `multiset`.

22.3.1 multiset Associative Container (Cont.)

- The purpose of p is to store the return value of multiset function equal_range that returns a pair containing the results of both a lower_bound and an upper_bound operation.
- Type pair contains two public data members called first and second.
- Line 57 uses function equal_range to determine the lower_bound and upper_bound of 22 in the multiset.
- Line 60 uses p.first and p.second, respectively, to access the lower_bound and upper_bound.
- We dereferenced the iterators to output the values at the locations returned from equal_range.

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

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

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

```

Fig. 22.20 | Standard Library `set` class template. (Part 1 of 3.)

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

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

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

   // insert 9.5 in doubleSet
   p = doubleSet.insert( 9.5 ); // value already in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );
   cout << endl;
} // end main
```

Fig. 22.20 | Standard Library `set` class template. (Part 2 of 3.)

```
doubleSet contains: 2.1 3.7 4.2 9.5

13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

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

Fig. 22.20 | Standard Library `set` class template. (Part 3 of 3.)

22.3.2 set Associative Container

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

22.3.2 set Associative Container (Cont.)

- Line 10 uses `typedef` to create a new type name (`DoubleSet`) for a set of `double` values ordered in ascending order, using the function object `less<double>`.
- Line 16 uses the new type `DoubleSet` to instantiate object `doubleSet`.
- The constructor call takes the elements in array `a` between `a` and `a + SIZE` (i.e., the entire array) and inserts them into the `set`.
- Line 20 uses algorithm `copy` to output the contents of the `set`.
- Notice that the value `2.1`—which appeared twice in array `a`—appears only once in `doubleSet`.

22.3.2 set Associative Container (Cont.)

- This is because container `set` does not allow duplicates.
- Line 23 defines a `pair` consisting of a `const_iterator` for a `DoubleSet` and a `bool` value.
- This object stores the result of a call to `set` function `insert`.
- Line 28 uses function `insert` to place the value `13.8` in the `set`.
- The returned `pair`, `p`, contains an iterator `p.first` pointing to the value `13.8` in the `set` and a `bool` value that is `true` if the value was inserted and `false` if the value was not inserted (because it was already in the `set`).
- In this case, `13.8` was not in the set, so it was inserted.
- Line 35 attempts to insert `9.5`, which is already in the set.
- The output of lines 36–37 shows that `9.5` was not inserted.

22.3.3 `multimap` Associative Container

- The `multimap` associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs).
- Many of the functions used with `multiset`s and `set`s are also used with `multimap`s and `map`s.
- The elements of `multimap`s and `map`s are `pair`s of keys and values instead of individual values.
- When inserting into a `multimap` or `map`, a `pair` object that contains the key and the value is used.
- The ordering of the keys is determined by a comparator function object.
- For example, in a `multimap` that uses integers as the key type, keys can be sorted in ascending order by ordering them with comparator function object `less< int >`.

Performance Tip 22.15

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

22.3.3 multimap Associative Container (Cont.)

- Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key.
- This is often called a one-to-many relationship.
- For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many students; in the military, one rank (like “private”) has many people.
- A `multimap` supports bidirectional iterators, but not random-access iterators.
- Figure 22.21 demonstrates the `multimap` associative container.
- Header file <map> must be included to use class `multimap`.

```cpp
// Fig. 22.21: Fig22_21.cpp
// Standard Library class multimap test program.
#include <iostream>
#include <map> // multimap class-template definition
using namespace std;

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

int main()
{
   Mmid pairs; // declare the multimap pairs

   cout << "There are currently " << pairs.count( 15
      << " pairs with key 15 in the multimap\n";

   // insert two value_type objects in pairs
   pairs.insert( Mmid::value_type( 15, 2.7 ) );
   pairs.insert( Mmid::value_type( 15, 99.3 ) );

```

Fig. 22.21 | Standard Library `multimap` class template. (Part 1 of 3.)

```
    cout << "After inserts, there are " << pairs.count( 15 )
       << " pairs with key 15\n\n";

    // insert five value_type objects in pairs
    pairs.insert( Mmid::value_type( 30, 111.11 ) );
    pairs.insert( Mmid::value_type( 10, 22.22 ) );
    pairs.insert( Mmid::value_type( 25, 33.333 ) );
    pairs.insert( Mmid::value_type( 20, 9.345 ) );
    pairs.insert( Mmid::value_type( 5, 77.54 ) );

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

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

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

Fig. 22.21 | Standard Library `multimap` class template. (Part 2 of 3.)

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

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

Fig. 22.21 | Standard Library `multimap` class template. (Part 3 of 3.)

22.3.3 `multimap` Associative Container (Cont.)

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

22.3.3 multimap Associative Container (Cont.)

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

22.3.3 `multimap` Associative Container (Cont.)

- Then lines 21–22 output the number of pairs with key `15`.
- Lines 25–29 insert five additional `pair`s into the `multimap`.
- The `for` statement in lines 34–36 outputs the contents of the `multimap`, including both keys and values.
- Line 36 uses the `const_iterator` called `iter` to access the members of the `pair` in each element of the `multimap`.
- Notice in the output that the keys appear in ascending order.

22.3.4 map Associative Container

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

```cpp
// Fig. 22.22: Fig22_22.cpp
// Standard Library class map test program.
#include <iostream>
#include <map> // map class-template definition
using namespace std;

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

int main()
{
   Mid pairs;

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

```

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

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

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

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

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

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

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

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

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

After subscript operations, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

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

22.3.4 map Associative Container (Cont.)

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

22.4 Container Adapters

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

22.4.1 stack Adapter

- Class `stack` enables insertions into and deletions from the underlying data structure at one end (commonly referred to as a last-in, first-out data structure).
- A `stack` can be implemented with any of the sequence containers: `vector`, `list` and `deque`.
- This example creates three integer stacks, using each of the sequence containers of the Standard Library as the underlying data structure to represent the `stack`.
- By default, a `stack` is implemented with a `deque`.

22.4.1 stack Adapter (Cont.)

- The `stack` operations are `push` to insert an element at the top of the `stack` (implemented by calling function `push_back` of the underlying container), `pop` to remove the top element of the `stack` (implemented by calling function `pop_back` of the underlying container), `top` to get a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `stack` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `stack` (implemented by calling function `size` of the underlying container).

Performance Tip 22.16

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

Performance Tip 22.17

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

22.4.1 stack Adapter (Cont.)

- Figure 22.23 demonstrates the `stack` adapter class.
- Header file <stack> must be included to use class `stack`.
- Lines 18, 21 and 24 instantiate three integer stacks.
- Line 18 specifies a `stack` of integers that uses the default `deque` container as its underlying data structure.
- Line 21 specifies a `stack` of integers that uses a `vector` of integers as its underlying data structure.

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

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

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

```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 1 of 4.)

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

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

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

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

```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 2 of 4.)

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

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

```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 3 of 4.)

```
// pop elements from stack object to which stackRef refers
template< typename T > void popElements( T &stackRef )
{
   while ( !stackRef.empty() )
   {
      cout << stackRef.top() << ' '; // view (and display) top element
      stackRef.pop(); // remove top element
   } // end while
} // end function popElements
```

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

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

Fig. 22.23 | Standard Library `stack` adapter class. (Part 4 of 4.)

22.4.1 stack Adapter (Cont.)

- Line 24 specifies a `stack` of integers that uses a `list` of integers as its underlying data structure.
- Function `pushElements` (lines 46–53) pushes the elements onto each `stack`.
- Line 50 uses function `push` (available in each adapter class) to place an integer on top of the `stack`.
- Line 51 uses `stack` function `top` to retrieve the top element of the `stack` for output.

22.4.1 stack Adapter (Cont.)

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

22.4.2 queue Adapter

- Class queue enables insertions at the back of the underlying data structure and deletions from the front (commonly referred to as a first-in, first-out data structure).
- A `queue` can be implemented with STL data structure `list` or `deque`.
- By default, a `queue` is implemented with a `deque`.

22.4.2 queue Adapter (Cont.)

- The common `queue` operations are `push` to insert an element at the back of the `queue` (implemented by calling function `push_back` of the underlying container), `pop` to remove the element at the front of the `queue` (implemented by calling function `pop_front` of the underlying container), front to get a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container), back to get a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `queue` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `queue` (implemented by calling function `size` of the underlying container).

Performance Tip 22.18

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

Performance Tip 22.19

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

22.4.2 queue Adapter (Cont.)

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

```
// Fig. 22.24: Fig22_24.cpp
// Standard Library adapter queue test program.
#include <iostream>
#include <queue> // queue adapter definition
using namespace std;

int main()
{
   queue< double > values; // queue with doubles

   // push elements onto queue values
   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";

```

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

```
   // pop elements from queue
   while ( !values.empty() )
   {
      cout << values.front() << ' '; // view front element
      values.pop(); // remove element
   } // end while

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

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

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

22.4.2 queue Adapter (Cont.)

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

22.4.3 priority_queue Adapter (Cont.)

- Class priority_queue provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure.
- A priority_queue can be implemented with STL sequence containers vector or deque.
- By default, a priority_queue is implemented with a vector as the underlying container.
- When elements are added to a priority_queue, they're inserted in priority order, such that the highest-priority element (i.e., the largest value) will be the first element removed from the priority_queue.

22.4.3 priority_queue Adapter (Cont.)

- This is usually accomplished by arranging the elements in a binary tree structure called a heap that always maintains the largest value (i.e., highest-priority element) at the front of the data structure.
- We discuss the STL's heap algorithms in Section 22.5.12.
- The comparison of elements is performed with comparator function object `less< T >` by default, but you can supply a different comparator.
- There are several common `priority_queue` operations.
- `push` inserts an element at the appropriate location based on priority order of the `priority_queue` (implemented by calling function `push_back` of the underlying container, then reordering the elements using heapsort).

22.4.3 priority_queue Adapter (Cont.)

- `pop` removes the highest-priority element of the `priority_queue` (implemented by calling function `pop_back` of the underlying container after removing the top element of the heap).
- `top` gets a reference to the top element of the `priority_queue` (implemented by calling function `front` of the underlying container).
- `empty` determines whether the `priority_queue` is empty (implemented by calling function `empty` of the underlying container).
- `size` gets the number of elements in the `priority_queue` (implemented by calling function `size` of the underlying container).

Performance Tip 22.20

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

Performance Tip 22.21

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

22.4.3 priority_queue Adapter (Cont.)

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

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

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

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

   cout << "Popping from priorities: ";

```

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

```
    // pop element from priority_queue
    while ( !priorities.empty() )
    {
       cout << priorities.top() << ' '; // view top element
       priorities.pop(); // remove top element
    } // end while

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

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

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

22.4.3 priority_queue Adapter (Cont.)

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

22.5 Algorithms

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

Software Engineering Observation 22.8

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

Software Engineering Observation 22.9

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

22.5.1 fill, fill_n, generate and generate_n

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

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

char nextLetter(); // prototype of generator function

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

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

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

```

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 1 of 3.)

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

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

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

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

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

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

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 2 of 3.)

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

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

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

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

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 3 of 3.)

22.5.1 `fill`, `fill_n`, `generate` and `generate_n` (Cont.)

- Line 13 defines a 10-element `vector` that stores `char` values.
- Line 15 uses function `fill` to place the character `'5'` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`.
- The iterators supplied as the first and second argument must be at least forward iterators (i.e., they can be used for both input from a container and output to a container in the forward direction).
- Line 21 uses function `fill_n` to place the character `'A'` in the first five elements of `vector chars`.

22.5.1 `fill`, `fill_n`, `generate` and `generate_n` (Cont.)

- The iterator supplied as the first argument must be at least an output iterator (i.e., it can be used for output to a container in the forward direction).
- The second argument specifies the number of elements to fill.
- The third argument specifies the value to place in each element.
- Line 27 uses function `generate` to place the result of a call to generator function `next-Letter` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`.

22.5.1 `fill`, `fill_n`, `generate` and `generate_n` (Cont.)

- The iterators supplied as the first and second arguments must be at least forward iterators.
- Function `nextLetter` (lines 42–46) begins with the character `'A'` maintained in a `static` local variable.
- The statement in line 45 postincrements the value of `letter` and returns the old value of `letter` each time `next-Letter` is called.
- Line 33 uses function `generate_n` to place the result of a call to generator function `nextLetter` in five elements of `vector chars`, starting from `chars.begin()`.
- The iterator supplied as the first argument must be at least an output iterator.

22.5.5 Mathematical Algorithms

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

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

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

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

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

```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 1 of 5.)

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

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

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

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

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

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

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 2 of 5.)

```

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

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

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

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

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

```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 3 of 5.)

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

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

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 4 of 5.)

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

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

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

The total of the elements in Vector v is: 55

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

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

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 5 of 5.)

22.5.5 Mathematical Algorithms (Cont.)

- Line 24 uses function random_shuffle to reorder randomly the elements in the range from `v.begin()` up to, but not including, `v.end()` in `v`.
- This function takes two random-access iterator arguments.
- Line 34 uses function count to count the elements with the value `8` in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`.
- This function requires its two iterator arguments to be at least input iterators.
- Line 38 uses function count_if to count elements in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2` for which the predicate function `greater9` returns `true`.

22.5.5 Mathematical Algorithms (Cont.)

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

22.5.5 Mathematical Algorithms (Cont.)

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

22.5.5 Mathematical Algorithms (Cont.)

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

22.5.5 Mathematical Algorithms (Cont.)

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

22.5.5 Mathematical Algorithms (Cont.)

- Line 60 uses function `transform` to apply a general function to every element in the range from `v.begin()` up to, but not including, `v.end()` in `v`.
- The general function (the fourth argument) should take the current element as an argument, should not modify the element and should return the `transform`ed value.
- Function `transform` requires its first two iterator arguments to be at least input iterators and its third argument to be at least an output iterator.
- The third argument specifies where the `transform`ed values should be placed.
- Note that the third argument can equal the first.

22.5.6 Basic Searching and Sorting Algorithms

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

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

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

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

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

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

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 1 of 4.)


```

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

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

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

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

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

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 2 of 4.)

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

   // use binary_search to locate 13 in v
   if ( binary_search( v.begin(), v.end(), 13 ) )
      cout << "\n\n13 was found in v";
   else
      cout << "\n\n13 was not found in v";

   // use binary_search to locate 100 in v
   if ( binary_search( v.begin(), v.end(), 100 ) )
      cout << "\n100 was found in v";
   else
      cout << "\n100 was not found in v";

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

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 3 of 4.)

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

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

Found 16 at location 4
100 not found

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

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

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

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 4 of 4.)

22.5.6 Basic Searching and Sorting Algorithms (Cont.)

- Line 23 uses function `find` to locate the value `16` in the range from `v.begin()` up to, but not including, `v.end()` in `v`.
- The function requires its two iterator arguments to be at least input iterators and returns an input iterator that either is positioned at the first element containing the value or indicates the end of the sequence (as is the case in line 31).
- Line 39 uses function `find_if` to locate the first value in the range from `v.begin()` up to, but not including, `v.end()` in `v` for which the unary predicate function `greater10` returns `true`.

22.5.6 Basic Searching and Sorting Algorithms (Cont.)

- Function `greater10` (defined in lines 71–74) takes an integer and returns a `bool` value indicating whether the integer argument is greater than 10.
- Function `find_if` requires its two iterator arguments to be at least input iterators.
- The function returns an input iterator that either is positioned at the first element containing a value for which the predicate function returns `true` or indicates the end of the sequence.
- Line 48 uses function `sort` to arrange the elements in the range from `v.begin()` up to, but not including, `v.end()` in `v` in ascending order.

22.5.6 Basic Searching and Sorting Algorithms (Cont.)

- The function requires its two iterator arguments to be random-access iterators.
- A second version of this function takes a third argument that is a binary predicate function taking two arguments that are values in the sequence and returning a `bool` indicating the sorting order—if the return value is `true`, the two elements being compared are in sorted order.

Common Programming Error 22.5

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

22.5.6 Basic Searching and Sorting Algorithms (Cont.)

- Line 53 uses function `binary_search` to determine whether the value 13 is in the range from `v.begin()` up to, but not including, `v.end()` in `v`.
- The sequence of values must be sorted in ascending order first.
- Function `binary_search` requires its two iterator arguments to be at least forward iterators.
- The function returns a `bool` indicating whether the value was found in the sequence.

22.5.6 Basic Searching and Sorting Algorithms (Cont.)

- Line 59 demonstrates a call to function `binary_search` in which the value is not found.
- A second version of this function takes a fourth argument that is a binary predicate function taking two arguments that are values in the sequence and returning a `bool`.
- The predicate function returns `true` if the two elements being compared are in sorted order.
- To obtain the location of the search key in the container, use the `lower_bound` or `find` algorithms.

22.5.7 swap, iter_swap and swap_ranges

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


```
// Fig. 22.32: Fig22_32.cpp
// Standard Library algorithms iter_swap, swap and swap_ranges.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <iterator>
using namespace std;

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

   cout << "Array a contains:\n   ";
   copy( a, a + SIZE, output ); // display array a

   // swap elements at locations 0 and 1 of array a
   swap( a[ 0 ], a[ 1 ] );

   cout << "\nArray a after swapping a[0] and a[1] using swap:\n   ";
   copy( a, a + SIZE, output ); // display array a

```

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

```cpp
    // use iterators to swap elements at locations 0 and 1 of array a
    iter_swap( &a[ 0 ], &a[ 1 ] ); // swap with iterators
    cout << "\nArray a after swapping a[0] and a[1] using iter_swap:\n   ";
    copy( a, a + SIZE, output );

    // swap elements in first five elements of array a with
    // elements in last five elements of array a
    swap_ranges( a, a + 5, a + 5 );

    cout << "\nArray a after swapping the first five elements\n"
         << "with the last five elements:\n   ";
    copy( a, a + SIZE, output );
    cout << endl;
 } // end main
```

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

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

22.5.7 swap, iter_swap and swap_ranges (Cont.)

- Line 24 uses function iter_swap to exchange the two elements.
- The function takes two forward iterator arguments (in this case, pointers to elements of an array) and exchanges the values in the elements to which the iterators refer.
- Line 30 uses function swap_ranges to exchange the elements from a up to, but not including, a + 5 with the elements beginning at position a + 5.

22.5.7 swap, iter_swap and swap_ranges (Cont.)

- The function requires three forward iterator arguments.
- The first two arguments specify the range of elements in the first sequence that will be exchanged with the elements in the second sequence starting from the iterator in the third argument.
- In this example, the two sequences of values are in the same array, but the sequences can be from different arrays or containers.

22.5.8 copy_backward, merge, unique and reverse

- Figure 22.33 demonstrates STL algorithms `copy_backward`, `merge`, `unique` and `reverse`.
- Line 26 uses function copy_backward to copy elements in the range from `v1.begin()` up to, but not including, `v1.end()`, placing the elements in `results` by starting from the element before `results.end()` and working toward the beginning of the `vector`.
- The function returns an iterator positioned at the last element copied into the `results` (i.e., the beginning of `results`, because of the backward copy).
- The elements are placed in `results` in the same order as `v1`.

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

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

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

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

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

```

Fig. 22.33 | Demonstrating copy_backward, merge, unique and reverse. (Part 1 of 3.)

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

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

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

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

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

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

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

```

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 2 of 3.)

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

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

After copy_backward, results contains: 1 3 5 7 9

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

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

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

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 3 of 3.)

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

- Line 33 uses function `merge` to combine two sorted ascending sequences of values into a third sorted ascending sequence.
- The function requires five iterator arguments.
- The first four must be at least input iterators and the last must be at least an output iterator.
- The first two arguments specify the range of elements in the first sorted sequence (`v1`), the second two arguments specify the range of elements in the second sorted sequence (`v2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged.

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

- A second version of this function takes as its sixth argument a binary predicate function that specifies the sorting order.
- Line 30 creates vector `results2` with the number of elements `v1.size()` + `v2.size()`.
- Using the `merge` function as shown here requires that the sequence where the results are stored be at least the size of the two sequences being merged.
- If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation, you can use the following statements:
  - ```
    vector< int > results2;
    merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
       back_inserter( results2 ) );
    ```
    ```

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

- The argument `back_inserter(results2)` uses function template back_in-serter (header file `<iterator>`) for the container `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*.
- Thus, the number of elements in the container does not have to be known in advance.

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

- There are two other inserters—front_inserter (to insert an element at the beginning of a container specified as its argument) and inserter (to insert an element before the iterator supplied as its second argument in the container supplied as its first argument).
- Line 40 uses function unique on the sorted sequence of elements in the range from results2.begin() up to, but not including, results2.end() in results2.
- After this function is applied to a sorted sequence with duplicate values, only a single copy of each value remains in the sequence.
- The function takes two arguments that must be at least forward iterators.

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

- The function returns an iterator positioned after the last element in the sequence of unique values.
- The values of all elements in the container after the last unique value are undefined.
- A second version of this function takes as a third argument a binary predicate function specifying how to compare two elements for equality.
- Line 46 uses function `reverse` to reverse all the elements in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1`.
- The function takes two arguments that must be at least bidirectional iterators.

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

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

```cpp
// Fig. 22.34: Fig22_34.cpp
// Standard Library algorithms inplace_merge,
// reverse_copy and unique_copy.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // back_inserter definition
using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE ); // copy of a
   ostream_iterator< int > output( cout, " " );

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

   // merge first half of v1 with second half of v1 such that
   // v1 contains sorted set of elements after merge
   inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 1 of 3.)

```

   cout << "\nAfter inplace_merge, v1 contains: ";
   copy( v1.begin(), v1.end(), output );

   vector< int > results1;

   // copy only unique elements of v1 into results1
   unique_copy( v1.begin(), v1.end(), back_inserter( results1 ) );
   cout << "\nAfter unique_copy results1 contains: ";
   copy( results1.begin(), results1.end(), output );

   vector< int > results2;

   // copy elements of v1 into results2 in reverse order
   reverse_copy( v1.begin(), v1.end(), back_inserter( results2 ) );
   cout << "\nAfter reverse_copy, results2 contains: ";
   copy( results2.begin(), results2.end(), output );
   cout << endl;
} // end main
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 2 of 3.)

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

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 3 of 3.)

22.5.9 inplace_merge, unique_copy and reverse_copy (Cont.)

- Line 30 uses function unique_copy to make a copy of all the unique elements in the sorted sequence of values from v1.begin() up to, but not including, v1.end().
- The copied elements are placed into vector results1.
- The first two arguments must be at least input iterators and the last must be at least an output iterator.
- In this example, we did not preallocate enough elements in results1 to store all the elements copied from v1.
- Instead, we use function back_inserter (defined in header file <iterator>) to add elements to the end of v1.

22.5.9 inplace_merge, unique_copy and reverse_copy (Cont.)

- The back_inserter uses class vector's capability to insert elements at the end of the vector.
- Because the back_inserter inserts an element rather than replacing an existing element's value, the vector is able to grow to accommodate additional elements.
- A second version of the unique_copy function takes as a fourth argument a binary predicate function for comparing elements for equality.

22.5.9 inplace_merge, unique_copy and reverse_copy (Cont.)

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

22.5.10 Set Operations

- Figure 22.35 demonstrates functions `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union` for manipulating sets of sorted values.
- To demonstrate that STL functions can be applied to arrays and containers, this example uses only arrays (remember, a pointer into an array is a random-access iterator).
- Lines 25 and 31 call function `includes`.
- Function `includes` compares two sets of sorted values to determine whether every element of the second set is in the first set.
- If so, `includes` returns `true`; otherwise, it returns `false`.

22.5.10 Set Operations (Cont.)

- The first two iterator arguments must be at least input iterators and must describe the first set of values.
- In line 25, the first set consists of the elements from `a1` up to, but not including, `a1 + SIZE1`.
- The last two iterator arguments must be at least input iterators and must describe the second set of values.
- In this example, the second set consists of the elements from `a2` up to, but not including, `a2 + SIZE2`.
- A second version of function `includes` takes a fifth argument that is a binary predicate function indicating the order in which the elements were originally sorted.
- The two sequences must be sorted using the same comparison function.

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

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

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

```

Fig. 22.35 | set operations of the Standard Library. (Part 1 of 4.)

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

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

   int difference[ SIZE1 ];

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

   int intersection[ SIZE1 ];

```

Fig. 22.35 | set operations of the Standard Library. (Part 2 of 4.)

```
   // determine elements in both a1 and a2
   ptr = set_intersection( a1, a1 + SIZE1,
      a2, a2 + SIZE2, intersection );
   cout << "\n\nset_intersection of a1 and a2 is: ";
   copy( intersection, ptr, output );

   int symmetric_difference[ SIZE1 + SIZE2 ];

   // determine elements of a1 that are not in a2 and
   // elements of a2 that are not in a1
   ptr = set_symmetric_difference( a1, a1 + SIZE1,
      a3, a3 + SIZE2, symmetric_difference );
   cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
   copy( symmetric_difference, ptr, output );

   int unionSet[ SIZE3 ];

   // determine elements that are in either or both sets
   ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
   cout << "\n\nset_union of a1 and a3 is: ";
   copy( unionSet, ptr, output );
   cout << endl;
} // end main
```

Fig. 22.35 | `set` operations of the Standard Library. (Part 3 of 4.)

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

a1 includes a2
a1 does not include a3

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

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

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

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

Fig. 22.35 | `set` operations of the Standard Library. (Part 4 of 4.)

22.5.10 Set Operations (Cont.)

- Lines 39–40 use function `set_difference` to find the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in ascending order).
- The elements that are different are copied into the fifth argument (in this case, the array `difference`).
- The first two iterator arguments must be at least input iterators for the first set of values.
- The next two iterator arguments must be at least input iterators for the second set of values.
- The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different.

22.5.10 Set Operations (Cont.)

- The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points.
- A second version of function `set_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted.
- The two sequences must be sorted using the same comparison function.
- Lines 47–48 use function `set_intersection` to determine the elements from the first set of sorted values that are in the second set of sorted values (both sets of values must be in ascending order).

22.5.10 Set Operations (Cont.)

- The elements common to both sets are copied into the fifth argument (in this case, array `intersection`).
- The first two iterator arguments must be at least input iterators for the first set of values.
- The next two iterator arguments must be at least input iterators for the second set of values.
- The fifth argument must be at least an output iterator indicating where to store a copy of the values that are the same.
- The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points.

22.5.10 Set Operations (Cont.)

- A second version of function `set_intersection` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted.
- The two sequences must be sorted using the same comparison function-.
- Lines 56–57 use function `set_symmetric_difference` to determine the elements in the first set that are not in the second set and the elements in the second set that are not in the first set (both sets must be in ascending order).
- The elements that are different are copied from both sets into the fifth argument (the array `symmetric_difference`).

22.5.10 Set Operations (Cont.)

- The first two iterator arguments must be at least input iterators for the first set of values.
- The next two iterator arguments must be at least input iterators for the second set of values.
- The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different.
- The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points.
- A second version of function `set_symmetric_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted.

22.5.10 Set Operations (Cont.)

- The two sequences must be sorted using the same comparison function.
- Line 64 uses function `set_union` to create a set of all the elements that are in either or both of the two sorted sets (both sets of values must be in ascending order).
- The elements are copied from both sets into the fifth argument (in this case the array `unionSet`).
- Elements that appear in both sets are only copied from the first set.
- The first two iterator arguments must be at least input iterators for the first set of values.

22.5.10 Set Operations (Cont.)

- The next two iterator arguments must be at least input iterators for the second set of values.
- The fifth argument must be at least an output iterator indicating where to store the copied elements.
- The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points.
- A second version of `set_union` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted.
- The two sequences must be sorted using the same comparison function.

22.5.13 `min` and `max`

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

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

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

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

Fig. 22.38 | Algorithms `min` and `max`.

22.5.14 STL Algorithms Not Covered in This Chapter

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

Algorithm	Description
inner_product	Calculate the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
adjacent_difference	Beginning with the second element in a sequence, calculate the difference (using operator −) between the current and previous elements, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
partial_sum	Calculate a running total (using operator +) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 1 of 5.)

Algorithm	Description
`nth_element`	Use three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this algorithm executes, all elements before the partitioning element are less than that element and all elements after the partitioning element are greater than or equal to that element. A second version of this algorithm takes as a fourth argument a binary comparison function.
`partition`	This algorithm is similar to `nth_element`, but requires less powerful bidirectional iterators, making it more flexible. It requires two bidirectional iterators indicating the range of elements to partition. The third argument is a unary predicate function that helps partition the elements so that all elements for which the predicate is `true` are to the left (toward the beginning of the sequence) of those for which the predicate is `false`. A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns `false`.
`stable_partition`	Similar to `partition` except that this algorithm guarantees that equivalent elements will be maintained in their original order.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 2 of 5.)

Algorithm	Description
`next_permutation`	Next lexicographical permutation of a sequence.
`prev_permutation`	Previous lexicographical permutation of a sequence.
`rotate`	Use three forward iterator arguments to rotate the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.
`rotate_copy`	This algorithm is identical to `rotate` except that the results are stored in a separate sequence indicated by the fourth argument—an output iterator. The two sequences must have the same number of elements.
`adjacent_find`	This algorithm returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the `end` of the sequence.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 3 of 5.)

Algorithm	Description
`search`	This algorithm searches for a subsequence of elements within a sequence of elements and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`search_n`	This algorithm searches a sequence of elements looking for a subsequence in which the values of a specified number of elements have a particular value and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`partial_sort`	Use three random-access iterators to sort part of a sequence. The first and last arguments indicate the sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The elements from the second argument iterator to the end of the sequence are in an undefined order.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 4 of 5.)

Algorithm	Description
`partial_sort_copy`	Use two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
`stable_sort`	The algorithm is similar to `sort` except that all equivalent elements are maintained in their original order. This sort is $O(n \log n)$ if enough memory is available; otherwise, it's $O(n(\log n)^2)$.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 5 of 5.)

22.6 Class bitset

- Class bitset makes it easy to create and manipulate bit sets, which are useful for representing a set of bit flags.
- bitsets are fixed in size at compile time.
- Class bitset is an alternate tool for bit manipulation, discussed in Chapter 21.
- The declaration
  - `bitset< size > b;`
- creates bitset b, in which every bit is initially 0.
- The statement
  - `b.set( bitNumber );`
- sets bit bitNumber of bitset b "on." The expression b.set() sets all bits in b "on."

22.6 Class bitset (Cont.)

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

22.6 Class bitset (Cont.)

- The statement
  - `b[ bitNumber ];`

  returns a reference to the bit `bitNumber` of `b`.
- Similarly,
  - `b.at( bitNumber );`

  performs range checking on `bitNumber` first.
  - If `bitNumber` is in range, `at` returns a reference to the bit.
  - Otherwise, `at` throws an `out_of_range` exception.

22.6 Class `bitset` (Cont.)

- The statement
  - `b.test( bitNumber );`

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

  returns the number of bits in `bitset` `b`.
- The expression
  - `b.count()`

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

22.6 Class `bitset` (Cont.)

- The expression
  - `b.any()`

  returns `true` if any bit is set in `bitset` `b`.
- The expression
  - `b.none()`

  returns `true` if none of the bits is set in `bitset` `b`.
- The expressions
  - `b == b1`
    `b != b1`

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

22.6 Class `bitset` (Cont.)

- Each of the bitwise assignment operators `&=`, `|=` and `^=` can be used to combine `bitset`s.
- For example,
  - `b &= b1;`

  `performs a bit-by-bit logical AND between bitsets b and b1.`
  - The result is stored in `b`.
- Bitwise logical OR and bitwise logical XOR are performed by
  - `b |= b1;`
    `b ^= b2;`
- The expression
  - `b >>= n;`

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

22.6 Class `bitset` (Cont.)

- The expression
  - `b <<= n;`

  shifts the bits in `bitset` `b` left by `n` positions.
- The expressions
  - `b.to_string()`
    `b.to_ulong()`

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

22.7 Function Objects

- Many STL algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task.
- For example, the `binary_search` algorithm that we discussed in Section 22.5.6 is overloaded with a version that requires as its fourth parameter a pointer to a function that takes two arguments and returns a `bool` value.
- The `binary_search` algorithm uses this function to compare the search key to an element in the collection.
- The function returns `true` if the search key and element being compared are equal; otherwise, the function returns `false`.

22.7 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.
- STL's designers made the algorithms more flexible by allowing any algorithm that can receive a function pointer to receive an object of a class that overloads the parentheses operator with a function named `operator()`, provided that the overloaded operator meets the requirements of the algorithm—in the case of `binary_search`, it must receive two arguments and return a `bool`.

22.7 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.
- Together, function objects and functions used are know as functors.
- Most algorithms can use function objects and functions interchangeably.

22.7 Function Objects (Cont.)

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

22.7 Function Objects (Cont.)

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

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

Fig. 22.41 | Function objects in the Standard Library.

Questions

?