Standard Template Library

Department of Software Engineering,

National University of Computer & Emerging Sciences,

Islamabad Campus

The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

The C++ Standard Template Libraries

- STL had three basic components:
  - Containers
    Generic class templates for storing collection of data.
  - Algorithms
    Generic function templates for operating on containers.
  - Iterators
    Generalized 'smart' pointers that facilitate use of containers.
    They provide an interface that is needed for STL algorithms to operate on STL containers.
- **String abstraction was added during standardization.**

Why use STL?

- STL offers an assortment of containers
- STL publicizes the time and storage complexity of its containers
- STL containers grow and shrink in size automatically
- STL provides built-in algorithms for processing containers
- STL provides iterators that make the containers and algorithms flexible and efficient.
- STL is extensible which means that users can add new containers and new algorithms such that:
  - STL algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well user defined containers

15.1 Introduction (Cont.)

Iterators

- Iterators, which have properties similar to those of *pointers*, are used to manipulate the container elements.

- Iterators encapsulate the mechanisms used to access container elements.
- This encapsulation enables many of the algorithms to be applied to various containers *independently* of the underlying container implementation.
- This also enables you to create new algorithms that can process the elements of *multiple* container types.

15.1 Introduction (Cont.)

Algorithms

- Standard Library algorithms are function templates that perform such common data manipulations as *searching*, *sorting* and *comparing elements (or entire containers).*
- The Standard Library provides many algorithms.
- Most of them use iterators to access container elements.
- Each algorithm has *minimum requirements* for the types of iterators that can be used with it.
- We'll see that containers support specific iterator types, some more powerful than others.
- A *container's* supported iterator type determines whether the container can be used with a specific algorithm.

15.2 Introduction to Containers

- The containers are divided into four major categories—sequence containers, ordered associative containers, unordered associative containers and container adapters.

Container class	Description
Sequence containers	
`array`	Fixed size. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`forward_list`	Singly linked list, rapid insertion and deletion anywhere. New in C++11.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
`vector`	Rapid insertions and deletions at back. Direct access to any element.
Ordered associative containers—keys are maintained in sorted order	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

Fig. 15.1 | Standard Library container classes and container adapters. (Part 1 of 2.)

Container class	Description
Unordered associative containers	
`unordered_set`	Rapid lookup, no duplicates allowed.
`unordered_multiset`	Rapid lookup, duplicates allowed.
`unordered_map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`unordered_multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 15.1 | Standard Library container classes and container adapters. (Part 2 of 2.)

15.2 Introduction to Containers (cont.)

Containers Overview

- The *sequence containers* represent *linear* data structures (i.e., all of their elements are conceptually "lined up in a row"), such as `array`s, `vector`s and linked lists.
- *Associative containers* are *nonlinear* data structures that typically can locate elements stored in the containers quickly.
- Such containers can store sets of values or key-value pairs.
- As of C++11, the keys in associative containers are *immutable* (they cannot be modified).

15.2 Introduction to Containers (cont.)

- The sequence containers and associative containers are collectively referred to as the first-class containers.
- Stacks and queues are typically constrained versions of sequence containers.
- For this reason, the Standard Library implements class templates `stack`s, `queue` and `priority_queue` as container adapters that enable a program to view a sequence container in a constrained manner.
- Class `string` supports the same functionality as a *sequence container*, but stores only character data.

15.2 Introduction to Containers (cont.)

Near Containers

- There are other container types that are considered near containers—built-in arrays, `bitset`s for maintaining sets of flag values and `valarrays` for performing high-speed *mathematical vector* (not to be confused with the `vector` container) operations.
- These types are considered *near containers* because they exhibit some, but not all, capabilities of the *first-class containers*.

15.2 Introduction to Containers (cont.)

Common Container Functions

- Most containers provide similar functionality.
- Many operations apply to all containers, and other operations apply to subsets of similar containers.
- Figure 15.2 describes the many functions that are commonly available in most Standard Library containers.
- Overloaded operators <, <=, >, >=, == and `!=` are not provided for `priority_queues`.

15.2 Introduction to Containers (cont.)

- Overloaded operators <, <=, >, >= are *not* provided for the *unordered associative containers*.
- Member functions `rbegin`, `rend`, `crbegin` and `crend` are not available in a `forward_list`.
- Before using any container, you should study its capabilities.

Member function	Description
default constructor	A constructor that *initializes an empty container*. Normally, each container has several constructors that provide different ways to initialize the container.
copy constructor	A constructor that initializes the container to be a *copy of an existing container* of the same type.
move constructor	A move constructor (new in C++11 and discussed in Chapter 24) moves the contents of an existing container of the same type into a new container. This avoids the overhead of copying each element of the argument container.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are *no* elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
copy `operator=`	Copies the elements of one container into another.

Fig. 15.2 | Common member functions for most Standard Library containers. (Part 1 of 4.)

Member function	Description
move `operator=`	The move assignment operator (new in C++11 and discussed in Chapter 24) moves the elements of one container into another. This avoids the overhead of copying each element of the argument container.
`operator<`	Returns `true` if the contents of the first container are *less than* the second; otherwise, returns `false`.
`operator<=`	Returns `true` if the contents of the first container are *less than or equal to* the second; otherwise, returns `false`.
`operator>`	Returns `true` if the contents of the first container are *greater than* the second; otherwise, returns `false`.
`operator>=`	Returns `true` if the contents of the first container are *greater than or equal to* the second; otherwise, returns `false`.
`operator==`	Returns `true` if the contents of the first container are *equal to* the contents of the second; otherwise, returns `false`.
`operator!=`	Returns `true` if the contents of the first container are *not equal to* the contents of the second; otherwise, returns `false`.

Fig. 15.2 | Common member functions for most Standard Library containers. (Part 2 of 4.)

Member function	Description
`swap`	Swaps the elements of two containers. As of C++11, there is now a non-member function version of swap that swaps the contents of its two arguments (which must be of the same container type) using move operations rather than copy operations.
`max_size`	Returns the *maximum number of elements* for a container.
`begin`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *first element* of the container.
`end`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *next position after the end* of the container.
`cbegin` (C++11)	Returns a `const_iterator` that refers to the container's *first element*.
`cend` (C++11)	Returns a `const_iterator` that refers to the *next position after the end* of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the *last element* of the container.

Fig. 15.2 | Common member functions for most Standard Library containers. (Part 3 of 4.)

Member function	Description
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the *position before the first element* of the container.
`crbegin` (C++11)	Returns a `const_reverse_iterator` that refers to the *last element* of the container.
`crend` (C++11)	Returns a `const_reverse_iterator` that refers to the *position before the first element* of the container.
`erase`	Removes *one or more* elements from the container.
`clear`	Removes *all* elements from the container.

Fig. 15.2 | Common member functions for most Standard Library containers. (Part 4 of 4.)

Strings

- In C we used char * to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string.

Hello World - C

```c
#include <stdio.h>

void main()
{
        // create string 'str' = "Hello world!"
        char *str = "Hello World!";

        printf("%s\n", str);
}
```

Hello World - C++

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        // create string 'str' = "Hello world!"
        string str = "Hello World!";

        cout << str << endl;
        return 0;
}
```

String

- To use the string type simply include its header file.

```
#include <string>
```

Creating strings

```
string str = "some text";
```

or

```
string str("some text");
```

other ways:

```
string s1(7,'a');
string s2 = s1;
```

string length

The length of string is returned by its `size()` operation.

```
#include <string>

string str = "something";
cout << "The size of "
     << str
     << "is " << str.size()
     << "characters." << endl;
```

The size method

str.size() ???

In C we had structs containing only data, In C++, we have :

```
class string
{
        …
 public:
        …
        unsigned int size();
        …
};
```

String concatenation

concatenating one string to another is done by the '+' operator.

```
string str1 = "Here ";
string str2 = "comes the sun";
string concat_str = str1 + str2;
```

String comparison

To check if two strings are equal use the '==' operator.

```
string str1 = "Here ";
string str2 = "comes the sun";

if ( str1 == str2 )
      /* do something */
else
      /* do something else */
```

String assignment

To assign one string to another use the "=" operator.

```
string str1 = "Sgt. Pappers";
string str2 = "lonely hearts club bend";
str2 = str1;
```

Now : str2 equals "Sgt. Pappers"

What more ?

- Containers
- Algorithms

Containers

Data structures that hold anything (other objects).

- list: doubly linked list.
- vector: similar to a C array, but dynamic.
- map: set of ordered key/value pairs.
- Set: set of ordered keys.

Algorithms

generic functions that handle common tasks such as searching, sorting, comparing, and editing:

- `find`
- `merge`
- `reverse`
- `sort`
- `and more: count, random shuffle, remove, Nth-element, rotate.`

Vector

- Provides an alternative to the built in array.
- A vector is self grown.
- Use It instead of the built in array!

Defining a new vector

Syntax: `vector<of what>`

For example :

`vector<int>` - vector of integers.
`vector<string>` - vector of strings.
`vector<int * >` - vector of pointers to integers.
`vector<Shape>` - vector of Shape objects. Shape is a user defined class.

Using Vector

- `#include <vector>`

- Two ways to use the vector type:
  1. Array style.
  2. STL style

Using a Vector – Array Style

We mimic the use of built-in array.

```
void simple_example()
{
        const int N = 10;
        vector<int> ivec(N);
        for (int i=0; i < 10; ++i)
                cin >> ivec[i];

        int ia[N];
        for ( int j = 0; j < N; ++j)
                ia[j] = ivec[j];
}
```

Using a vector - STL style

We define an empty vector

```
vector<string> svec;
```

we insert elements into the vector using the method push_back.

```
string word;
while ( cin >> word ) //the number of
                      words is unlimited.
{
      svec.push_back(word);
}
```

Insertion

```
void push_back(const T& x);
```

Inserts an element with value x at the end of the controlled sequence.

```
svec.push_back(str);
```

Size

```
unsigned int size();
```

Returns the length of the controlled sequence (how many items it contains).

```
unsigned int size = svec.size();
```

Vector Sequence Container

In STL, vectors are a type of container that provides a dynamic array that can grow or shrink in size at runtime. Vectors are implemented as an array that is dynamically resized when elements are added or removed. Vectors provide random access to elements, which means you can access elements using an index.

To use vectors in C++, you first need to include the <vector> header file. Here is an example of creating and using a vector that stores integers:

Example

```cpp
#include <vector>
#include <iostream>

using namespace std;

int main() {
    vector<int> myVector;

    // Add elements to the vector
    myVector.push_back(10);
    myVector.push_back(20);
    myVector.push_back(30);

    // Access elements using an index
    cout << "Element at index 0: " << myVector[0] << endl;

    // Iterate over the elements using an iterator
    vector<int>::iterator it;
    for (it = myVector.begin(); it != myVector.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
```

Example

```
    cout << "Vector size: " << myVector.size() << endl;

    // Remove an element from the vector
    myVector.pop_back();

    // Get the new size of the vector
    cout << "Vector size after pop_back(): " << myVector.size() << endl;

    return 0;
}
```

Vector Functions

- Vectors provide a wide range of member functions for accessing and modifying the contents of the vector, including:

- push_back(): Adds an element to the end of the vector.
- pop_back(): Removes the last element from the vector.
- size(): Returns the number of elements in the vector.
- empty(): Returns true if the vector is empty.
- front(): Returns a reference to the first element in the vector.
- back(): Returns a reference to the last element in the vector.
- insert(): Inserts an element at a specified position in the vector.
- erase(): Removes an element from the vector at a specified position.
- clear(): Removes all elements from the vector.
- Vectors provide a fast and efficient way to store and manipulate collections of elements in C++.

Vector Functions

- vector<Type> v - create an empty vector of type Type

```
vector<int> v1; // creates an empty integer vector
vector<string> v2; // creates an empty string vector
```

Vector Functions

- vector<Type> v(n, value) - create a vector of type Type with n elements, all initialized to value

- vector<int> v1(5, 10); // creates a vector with 5 elements, all initialized to 10
- vector<string> v2(3, "hello"); // creates a vector with 3 elements, all initialized to "hello"

Vector Functions

- v.push_back(value) - add an element value to the end of the vector v

- vector<int> v;
- v.push_back(10); // adds 10 to the end of the vector
- v.push_back(20); // adds 20 to the end of the vector

Vector Functions

- v.pop_back() - remove the last element from the vector v

- vector<int> v = {10, 20, 30};
- v.pop_back(); // removes 30 from the end of the vector

Vector Functions

- v.size() - return the number of elements in the vector v

- vector<int> v = {10, 20, 30};
- int size = v.size(); // size is 3

Vector Functions

- v.empty() - return true if the vector v is empty, false otherwise

- vector<int> v1;
- bool empty1 = v1.empty(); // empty1 is true

- vector<int> v2 = {10, 20, 30};
- bool empty2 = v2.empty(); // empty2 is false

Vector Functions

- v.clear() - remove all elements from the vector v

- vector<int> v = {10, 20, 30};
- v.clear(); // removes all elements from the vector

Vector Functions

- v.front() - return a reference to the first element in the vector v

- vector<int> v = {10, 20, 30};
- int& first = v.front(); // first is a reference to 10

Vector Functions

- v.back() - return a reference to the last element in the vector v

- v.back() - return a reference to the last element in the vector v

Vector Functions

- v.at(index) - return a reference to the element at the specified index in the vector v, with bounds checking

- vector<int> v = {10, 20, 30};
- int& second = v.at(1); // second is a reference to 20
- int& invalid = v.at(3); // throws an out_of_range exception

Vector Functions

- v.begin() - return an iterator to the beginning of the vector v

- vector<int> v = {10, 20, 30};
- vector<int>::iterator it = v.begin(); // it points to the first element of the vector

- v.end() - return an iterator to the end of the vector v

Insert at any Index

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> myVector{10, 20, 30};

    // Insert an element at index 1
    std::vector<int>::iterator it = myVector.begin() + 1;
    myVector.insert(it, 15);

    // Insert multiple elements at index 2
    it = myVector.begin() + 2;
    myVector.insert(it, {25, 35});

    // Print the vector
    for (std::vector<int>::size_type i = 0; i < myVector.size(); i++) {
        std::cout << myVector[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Remove from any Index

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> myVector{10, 20, 30, 40, 50};

    // Remove the element at index 2
    std::vector<int>::iterator it = myVector.begin() + 2;
    myVector.erase(it);

    // Print the vector
    for (std::vector<int>::size_type i = 0; i < myVector.size(); i++) {
        std::cout << myVector[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Class Exercise 1

Write a program that read integers from the user, sorts them, and print the result.

Solving the problem

- Easy way to read input.
- A "place" to store the input
- A way to sort the stored input.

Using STL

```
int main()
{
        int input;
        vector<int> ivec;

        /* rest of code */
}
```

STL - Input

```
while ( cin >> input )
      ivec.push_back(input);
```

STL - Sorting

```
sort(ivec.begin(), ivec.end());
```

Sort Prototype:

```
void  sort(Iterator first, Iterator  last);
```

STL - Output

```
for ( int i = 0; i < ivec.size(); ++i )
      cout << ivec[i] << " ";
cout << endl;
```

Or (more recommended)

```
vector<int>::iterator it;
for ( it = ivec.begin(); it != ivec.end(); ++it )
      cout << *it << " ";
cout << endl;
```

STL - Include files

```
#include <iostream>   // I/O
#include <vector>     // container
#include <algorithm> // sorting

//using namespace std;
```

Putting it all together

```
int main() {
        int input;
        vector<int> ivec;

        // input
        while (cin >> input )
                ivec.push_back(input);

        // sorting
        sort(ivec.begin(), ivec.end());

        // output
        vector<int>::iterator it;
        for ( it = ivec.begin();
              it != ivec.end(); ++it ) {
                    cout << *it << " ";
        }
        cout << endl;

        return 0;
}
```

Operations on vector

- iterator **begin**();
- iterator **end**();
- bool **empty**();
- void **push_back**(const T& x);
- iterator **erase**(iterator it);
- iterator **erase**(iterator first, iterator last);
- void **clear**();
- ….

15.5.2 `list` Sequence Container

- The `list` *sequence container* (from header `<list>`) allows 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 15.5.3) 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.

15.5.2 `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.

15.5.2 `list` Sequence Container (Cont.)

C++11: `forward_list` *Container*

- C++11 now includes the new `forward_list` sequence container (header `<forward_list>`), which is implemented as a *singly linked list*—every node in the list contains a pointer to the next node in the list.
- This enables class template list to support *forward iterators* that allow the container to be traversed in the forward direction.
- Any algorithm that requires *input*, *output* or *forward iterators* can operate on a `forward_list`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Functions

- In addition to the member functions in Fig. 15.2 and the common member functions of all *sequence containers* discussed in Section 15.5, class template `list` provides nine other member functions—including `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`.

15.5.2 `list` Sequence Container (Cont.)

- Several of these member functions are `list`-optimized implementations of Standard Library algorithms presented in Chapter 16.
- Both `push_front` and `pop_front` are also supported by `forward_list` and `deque`.
- Figure 15.13 demonstrates several features of class `list`.
- Remember that many of the functions presented in Figs. 15.10–15.11 can be used with class `list`, so we focus on the new features in this example's discussion.

```
1  // Fig. 15.13: fig15_13.cpp
2  // Standard library list class template.
3  #include <iostream>
4  #include <array>
5  #include <list> // list class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 // prototype for function template printList
11 template < typename T > void printList( const list< T > &listRef );
12
13 int main()
14 {
15    const size_t SIZE = 4;
16    array< int, SIZE > ints = { 2, 6, 4, 8 };
17    list< int > values; // create list of ints
18    list< int > otherValues; // create list of ints
19
20    // insert items in values
21    values.push_front( 1 );
22    values.push_front( 2 );
23    values.push_back( 4 );
24    values.push_back( 3 );
```

Fig. 15.13 | Standard Library `list` class template. (Part 1 of 6.)

```
25
26      cout << "values contains: ";
27      printList( values );
28
29      values.sort(); // sort values
30      cout << "\nvalues after sorting contains: ";
31      printList( values );
32
33      // insert elements of ints into otherValues
34      otherValues.insert( otherValues.cbegin(), ints.cbegin(), ints.cend() );
35      cout << "\nAfter insert, otherValues contains: ";
36      printList( otherValues );
37
38      // remove otherValues elements and insert at end of values
39      values.splice( values.cend(), otherValues );
40      cout << "\nAfter splice, values contains: ";
41      printList( values );
42
43      values.sort(); // sort values
44      cout << "\nAfter sort, values contains: ";
45      printList( values );
46
```

Fig. 15.13 | Standard Library `list` class template. (Part 2 of 6.)

```
47      // insert elements of ints into otherValues
48      otherValues.insert( otherValues.cbegin(), ints.cbegin(), ints.cend() );
49      otherValues.sort(); // sort the list
50      cout << "\nAfter insert and sort, otherValues contains: ";
51      printList( otherValues );
52
53      // remove otherValues elements and insert into values in sorted order
54      values.merge( otherValues );
55      cout << "\nAfter merge:\n   values contains: ";
56      printList( values );
57      cout << "\n   otherValues contains: ";
58      printList( otherValues );
59
60      values.pop_front(); // remove element from front
61      values.pop_back(); // remove element from back
62      cout << "\nAfter pop_front and pop_back:\n   values contains: "
63      printList( values );
64
65      values.unique(); // remove duplicate elements
66      cout << "\nAfter unique, values contains: ";
67      printList( values );
68
```

Fig. 15.13 | Standard Library `list` class template. (Part 3 of 6.)

```
69      // swap elements of values and otherValues
70      values.swap( otherValues );
71      cout << "\nAfter swap:\n   values contains: ";
72      printList( values );
73      cout << "\n   otherValues contains: ";
74      printList( otherValues );
75
76      // replace contents of values with elements of otherValues
77      values.assign( otherValues.cbegin(), otherValues.cend() );
78      cout << "\nAfter assign, values contains: ";
79      printList( values );
80
81      // remove otherValues elements and insert into values in sorted order
82      values.merge( otherValues );
83      cout << "\nAfter merge, values contains: ";
84      printList( values );
85
86      values.remove( 4 ); // remove all 4s
87      cout << "\nAfter remove( 4 ), values contains: ";
88      printList( values );
89      cout << endl;
90  } // end main
91
```

Fig. 15.13 | Standard Library `list` class template. (Part 4 of 6.)

```
92  // printList function template definition; uses
93  // ostream_iterator and copy algorithm to output list elements
94  template < typename T > void printList( const list< T > &listRef )
95  {
96     if ( listRef.empty() ) // list is empty
97        cout << "List is empty";
98     else
99     {
100       ostream_iterator< T > output( cout, " " );
101       copy( listRef.cbegin(), listRef.cend(), output );
102    } // end else
103 } // end function printList
```

Fig. 15.13 | Standard Library `list` class template. (Part 5 of 6.)

```
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
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8r
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. 15.13 | Standard Library `list` class template. (Part 6 of 6.)

15.5.2 `list` Sequence Container (Cont.)

Creating `list` Objects

- Lines 17–18 instantiate two `list` objects capable of storing `int`s.
- Lines 21–22 use function push_front to insert integers at the beginning of `values`.
- Function `push_front` is specific to classes `forward_list`, `list` and `deque`.
- Lines 23–24 use function `push_back` to insert integers at the end of `values`.
- *Function `push_back` is common to all sequence containers*, except `array` and `forward_list`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `sort`

- Line 29 uses `list` member function sort to arrange the elements in the `list` in *ascending order*.
- 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 whether the first argument should come before the second in the sorted contents.
- This function determines the order in which the elements of the `list` are sorted.

15.5.2 `list` Sequence Container (Cont.)

- This version could be particularly useful for a `list` that stores pointers rather than values.
- [*Note: We demonstrate a unary predicate function in Fig. 16.3.*]
- A unary predicate function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.]

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `splice`

- Line 39 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.

15.5.2 `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.
- Class template `forward_list` provides a similar member function named `splice_after`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `merge`

- After inserting more elements in `othervalues` and sorting both `values` and `other-values`, line 54 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 *binary predicate function* that takes two arguments (values in the list) and returns a `bool` value.
- The predicate function specifies the sorting order used by `merge`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `pop_front`

- Line 60 uses `list` function pop_front to remove the first element in the `list`.
- Line 60 uses function pop_back (available for *sequence containers* other than `array` and `forward_list`) to remove the last element in the `list`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `unique`

- Line 65 uses `list` function unique to *remove duplicate elements* in the `list`.
- The `list` should be in *sorted* order (so that all duplicates are side by side) before this operation is performed, to guarantee that all duplicates are eliminated.
- A second version of `unique` enables you to supply a *predicate function* that takes two arguments (values in the list) and returns a `bool` value specifying whether two elements are equal.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Function `swap`

- Line 70 uses function swap (available to all *first-class containers*) to exchange the contents of `values` with the contents of `othervalues`.

15.5.2 `list` Sequence Container (Cont.)

`list` Member Functions `assign` and `remove`

- Line 77 uses `list` function assign (available to all *sequence containers*) to replace the contents of `values` with the contents of `otherValues` in the range specified by the two iterator arguments.
- A second version of `assign` replaces the original contents with copies of the value specified in the second argument.
- The first argument of the function specifies the number of copies.
- Line 86 uses `list` function remove to delete all copies of the value `4` from the `list`.

15.5.3 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 Standard Library algorithms.

15.5.3 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 15.7.2).
- Additional storage for a `deque` can be allocated at either end of the `deque` in blocks of memory that are typically maintained as a built-in 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, `array`s or built-in arrays.

15.5.3 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 15.14 demonstrates features of class `deque`.
- Remember that many of the functions presented in Fig. 15.10, Fig. 15.11 and Fig. 15.13 also can be used with class `deque`.
- Header `<deque>` must be included to use class `deque`.

```
1  // Fig. 15.14: fig15_14.cpp
2  // Standard Library deque class template.
3  #include <iostream>
4  #include <deque> // deque class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    deque< double > values; // create deque of doubles
12    ostream_iterator< double > output( cout, " " );
13
14    // insert elements in values
15    values.push_front( 2.2 );
16    values.push_front( 3.5 );
17    values.push_back( 1.1 );
18
19    cout << "values contains: ";
20
21    // use subscript operator to obtain elements of values
22    for ( size_t i = 0; i < values.size(); ++i )
23       cout << values[ i ] << ' ';
24
```

Fig. 15.14 | Standard Library deque class template. (Part 1 of 2.)

```
25      values.pop_front(); // remove first element
26      cout << "\nAfter pop_front, values contains: ";
27      copy( values.cbegin(), values.cend(), output );
28
29      // use subscript operator to modify element at location 1
30      values[ 1 ] = 5.4;
31      cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
32      copy( values.cbegin(), values.cend(), output );
33      cout << endl;
34   } // end main
```

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

Fig. 15.14 | Standard Library deque class template. (Part 2 of 2.)

15.5.3 deque Sequence Container (Cont.)

- 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`.
- 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`.

15.5.3 deque Sequence Container (Cont.)

- Line 25 uses function `pop_front` to demonstrate removing the first element of the `deque`.
- Line 30 uses the subscript operator to obtain an *lvalue*.
- This enables values to be assigned directly to any element of the `deque`.

15.6 Associative Containers

- The *associative containers* provide *direct access* to store and retrieve elements via keys (often called search keys).
- The four *ordered associative containers* are `multiset`, `set`, `multimap` and `map`.
- Each of these maintains its keys in sorted order.
- There are also four corresponding *unordered associative containers*—`unordered_multiset`, `unordered_set`, `unordered_multimap` and `unordered_map`—that offer the most of the same capabilities as their ordered counterparts.
- The primary difference between the ordered and unordered associative containers is that the unordered ones do not maintain their keys in sorted order. In this section, we focus on the ordered associative containers.
-

15.6 Associative Containers (cont.)

- Iterating through an *ordered 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.

15.6 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 container member functions, *ordered associative containers* also support several other member functions that are specific to associative containers.
- Examples of each of the *ordered associative containers* and their common member functions are presented in the next several subsections.

15.6.1 `multiset` Associative Container

- The `multiset` *ordered associative container* (from header `<set>`) provides fast storage and retrieval of keys and allows duplicate keys.
- The elements' ordering is determined by a so-called 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 16.4.
- The data type of the keys in all *ordered associative containers* must support comparison based on the comparator function object—keys sorted with `less<T>` must support comparison with `operator<`.

15.6.1 `multiset` Associative Container (Cont.)

- If the keys used in the *ordered 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*).
- In if the order of the keys is not important, you can use `unordered_multiset` (header `<unordered_set>`) instead.
- Figure 15.15 demonstrates the `multiset` *ordered associative container* for a `multiset` of `int`s with keys that are sorted in *ascending order*.
- Containers `multiset` and `set` (Section 15.6.2) provide the same basic functionality.

```
1  // Fig. 15.15: fig15_15.cpp
2  // Standard Library multiset class template
3  #include <array>
4  #include <iostream>
5  #include <set> // multiset class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t SIZE = 10;
13    array< int, SIZE > a = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
14    multiset< int, less< int > > intMultiset; // multiset of ints
15    ostream_iterator< int > output( cout, " " );
16
17    cout << "There are currently " << intMultiset.count( 15 )
18       << " values of 15 in the multiset\n";
19
20    intMultiset.insert( 15 ); // insert 15 in intMultiset
21    intMultiset.insert( 15 ); // insert 15 in intMultiset
22    cout << "After inserts, there are " << intMultiset.count( 15 )
23       << " values of 15 in the multiset\n\n";
24
```

Fig. 15.15 | Standard Library `multiset` class template. (Part 1 of 3.)

```
25      // find 15 in intMultiset; find returns iterator
26      auto result = intMultiset.find( 15 );
27
28      if ( result != intMultiset.end() ) // if iterator not at end
29         cout << "Found value 15\n"; // found search value 15
30
31      // find 20 in intMultiset; find returns iterator
32      result = intMultiset.find( 20 );
33
34      if ( result == intMultiset.end() ) // will be true hence
35         cout << "Did not find value 20\n"; // did not find 20
36
37      // insert elements of array a into intMultiset
38      intMultiset.insert( a.cbegin(), a.cend() );
39      cout << "\nAfter insert, intMultiset contains:\n";
40      copy( intMultiset.begin(), intMultiset.end(), output );
41
42      // determine lower and upper bound of 22 in intMultiset
43      cout << "\n\nLower bound of 22: "
44         << *( intMultiset.lower_bound( 22 ) );
45      cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );
46
```

Fig. 15.15 | Standard Library `multiset` class template. (Part 2 of 3.)

```
47      // use equal_range to determine lower and upper bound
48      // of 22 in intMultiset
49      auto p = intMultiset.equal_range( 22 );
50
51      cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
52         << *( p.first ) << "\n   Upper bound: " << *( p.second );
53      cout << endl;
54   } // end main
```

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

15.6.1 `multiset` Associative Container (Cont.)

Creating a `multiset`

- Line 14 creates a `multiset` of `int`s ordered in *ascending order*, using the function object less<int>.
- *Ascending order* is the default for a `multiset`, so `less<int>` can be omitted.
- C++11 fixes a compiler issue with spacing between the closing > of `less<int>` and the closing > of the `multiset` type.

15.6.1 `multiset` Associative Container (Cont.)

- Before C++11, if you specified this multiset's type as
  ```
  multiset<int, less<int>> intMultiset;
  ```
- the compiler would treat `>>` at the end of the type as the `>>` operator and generate a compilation error.
- For this reason, you were required to put a space between the closing `>` of `less<int>` and the closing `>` of the `multiset` type (or any other similar template type, such as `vector<vector<int>>`).
- As of C++11, the preceding declaration compiles correctly.

15.6.1 `multiset` Associative Container (Cont.)

`multiset` Member Function `count`

- Line 17 uses function count (available to all *associative containers*) to count the number of occurrences of the value `15` currently in the `multiset`.

15.6.1 `multiset` Associative Container (Cont.)

`multiset` Member Function `insert`

- Lines 20–21 use one of the several overloaded 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.

15.6.1 multiset Associative Container (Cont.)

multiset Member Function find

- Line 26 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 calling end on the container.
- Line 32 demonstrates this case.

15.6.1 `multiset` Associative Container (Cont.)

Inserting Elements of Another Container into a `multiset`

- Line 38 uses function insert to insert the elements of array `a` into the `multiset`.
- In line 40, the `copy` algorithm copies the elements of the `multiset` to the standard output in *ascending order*.

15.6.1 `multiset` Associative Container (Cont.)

`multiset` Member Functions `lower_bound` and `upper_bound`

- Lines 44 and 45 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`.

15.6.1 `multiset` Associative Container (Cont.)

`pair` Objects and `multiset` Member Function `equal_range`

- Line 49 creates and intializes a `pair` object called `p`.
- Once again, we use C++11's `auto` keyword to infer the variable's type from its initializer—in this case, the return value of `multiset` member function `equal_range`, which is a `pair` object.
- Such objects associate pairs of values.
- The contents of a `p` will be two `const_iterators` for our `multiset` of `int`s.

15.6.1 `multiset` Associative Container (Cont.)

- The `multiset` function `equal_range` returns a pair containing the results of calling both `lower_bound` and `upper_bound`.
- Type `pair` contains two `public` data members called `first` and `second`. Line 49 uses function `equal_range` to determine the `lower_bound` and `upper_bound` of 22 in the `multiset`.
- Line 52 uses `p.first` and `p.second` to access the `lower_bound` and `upper_bound`.
- We *dereferenced* the iterators to output the values at the locations returned from `equal_range`.
- Though we did not do so here, you should always ensure that the iterators returned by `lower_bound`, `upper_bound` and `equal_range` are not equal to the container's end iterator before dereferencing the iterators.

15.6.1 `multiset` Associative Container (Cont.)

C++11: Variadic Class Template `tuple`

- C++ also includes class template `tuple`, which is similar to `pair`, but can hold any number of items of various types.
- As of C++11, class template `tuple` has been reimplemented using *variadic templates*—templates that can receive a *variable* number of arguments.
- We discuss `tuple` and variadic templates in Chapter 24, C++11: Additional Features.

15.6.2 `set` Associative Container

- The `set` *associative container* (from header `<set>`) 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 15.16 demonstrates a `set` of `double`s.

```
1  // Fig. 15.16: fig15_16.cpp
2  // Standard Library set class template.
3  #include <iostream>
4  #include <array>
5  #include <set>
6  #include <algorithm>
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 int main()
11 {
12    const size_t SIZE = 5;
13    array< double, SIZE > a = { 2.1, 4.2, 9.5, 2.1, 3.7 };
14    set< double, less< double > > doubleSet( a.begin(), a.end() );
15    ostream_iterator< double > output( cout, " " );
16
17    cout << "doubleSet contains: ";
18    copy( doubleSet.begin(), doubleSet.end(), output );
19
```

Fig. 15.16 | Standard Library `set` class template. (Part 1 of 3.)

```
20      // insert 13.8 in doubleSet; insert returns pair in which
21      // p.first represents location of 13.8 in doubleSet and
22      // p.second represents whether 13.8 was inserted
23      auto p = doubleSet.insert( 13.8 ); // value not in set
24      cout << "\n\n" << *( p.first )
25         << ( p.second ? " was" : " was not" ) << " inserted";
26      cout << "\ndoubleSet contains: ";
27      copy( doubleSet.begin(), doubleSet.end(), output );
28
29      // insert 9.5 in doubleSet
30      p = doubleSet.insert( 9.5 ); // value already in set
31      cout << "\n\n" << *( p.first )
32         << ( p.second ? " was" : " was not" ) << " inserted";
33      cout << "\ndoubleSet contains: ";
34      copy( doubleSet.begin(), doubleSet.end(), output );
35      cout << endl;
36   } // end main
```

Fig. 15.16 | 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. 15.16 | Standard Library `set` class template. (Part 3 of 3.)

15.6.2 `set` Associative Container (Cont.)

- Line 14 creates a `set` of `double`s ordered in *ascending order*, using the function object `less<double>`.
- The constructor call takes all the elements in `array` and inserts them into the `set`.
- Line 18 uses algorithm `copy` to output the contents of the `set`.
- Notice that the value `2.1`—which appeared twice in `array`—appears only *once* in `doubleSet`.
- This is because container `set` does *not* allow duplicates.

15.6.2 `set` Associative Container (Cont.)

- Line 23 defines and initializes a `pair` to store the result of a call to `set` function `insert`.
- The `pair` returned consists of a `const_iterator` pointing to the item in the `set` inserted and a `bool` value indicating whether the item was inserted—`true` if the item was not in the `set`; `false` if it was.
- Line 23 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` because the value was inserted.
- Line 30 attempts to insert `9.5`, which is already in the `set`.
- The output of shows that `9.5` was not inserted again because `set`s don't allow duplicate keys.
- In this case, `p.first` in the returned `pair` points to the existing 9.5 in the `set`.

15.6.3 `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 >`.

15.6.3 `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 15.17 demonstrates the *`multimap` associative container*.
- Header `<map>` must be included to use class `multimap`.
- If the order of the keys is not important, you can use `unordered_multimap` (header `<unordered_map>`) instead.

```
1  // Fig. 15.17: fig15_17.cpp
2  // Standard Library multimap class template.
3  #include <iostream>
4  #include <map> // multimap class-template definition
5  using namespace std;
6
7  int main()
8  {
9     multimap< int, double, less< int > > pairs; // create multimap
10
11    cout << "There are currently " << pairs.count( 15
12       << " pairs with key 15 in the multimap\n";
13
14    // insert two value_type objects in pairs
15    pairs.insert( make_pair( 15, 2.7 ) );
16    pairs.insert( make_pair( 15, 99.3 ) );
17
18    cout << "After inserts, there are " << pairs.count( 15 )
19       << " pairs with key 15\n\n";
20
```

Fig. 15.17 | Standard Library `multimap` class template. (Part 1 of 3.)

```
21      // insert five value_type objects in pairs
22      pairs.insert( make_pair( 30, 111.11 ) );
23      pairs.insert( make_pair( 10, 22.22 ) );
24      pairs.insert( make_pair( 25, 33.333 ) );
25      pairs.insert( make_pair( 20, 9.345 ) );
26      pairs.insert( make_pair( 5, 77.54 ) );
27
28      cout << "Multimap pairs contains:\nKey\tValue\n";
29
30      // walk through elements of pairs
31      for ( auto mapItem : pairs )
32         cout << mapItem.first << '\t' << mapItem.second << '\n';
33
34      cout << endl;
35   } // end main
```

Fig. 15.17 | 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. 15.17 | Standard Library `multimap` class template. (Part 3 of 3.)

15.6.3 `multimap` Associative Container (Cont.)

- Line 9 creates a `multimap` 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 11 uses function `count` to determine the number of key-value pairs with a key of `15` (none yet, since the container is currently empty).

15.6.3 `multimap` Associative Container (Cont.)

- Line 15 uses function `insert` to add a new key-value pair to the `multimap`.
- The expression `make_pair ( 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`.
- Function `make_pair` automatically uses the types that you specified for the keys and values in the `multimap`'s declaration (line 9).
- Line 16 inserts another `pair` object with the key `15` and the value `99.3`.

15.6.3 `multimap` Associative Container (Cont.)

- Then lines 18–19 output the number of pairs with key `15`.
- As of C++11, you can use list initialization for `pair` objects, so line 15 can be simplified as

```
pairs.insert( { 15, 2.7 } );
```

- Similarly, C++11 enables you to use list initialization to initialize an object being returned from a function.
- For example, if a function returns a `pair` containing an `int` and a `double`, you could write:

```
return { 15, 2.7 };
```

15.6.3 `multimap` Associative Container (Cont.)

- Lines 22–26 insert five additional `pair`s into the `multimap`.
- The range-based `for` statement in lines 31–32 outputs the contents of the `multimap`, including both keys and values.
- We infer the type of the loop's control variable (a `pair` containing an `int` key and a `double` value) with keyword `auto`.
- Line 32 accesses the members of the current `pair` in each element of the `multimap`.
- Notice in the output that the keys appear in *ascending order*.

15.6.3 multimap Associative Container (Cont.)

- ***C++11: List Initializing a Key–Value Pair Container***
- In this example, we used separate calls to member function insert to place key–value pairs in a `multimap`.
- If you know the key–value pairs in advance, you can use list initialization when you create the `multimap`.
- For example, the following statement initializes a `multimap` with three key–value pairs that are represented by the sublists in the main intializer list:

```
multimap< int, double, less< int > > pairs =
   { { 10, 22.22 }, { 20, 9.345 }, { 5, 77.54 } };
```

15.6.4 `map` Associative Container

- The *`map` associative container* (from header `<map>`) 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.
- Providing the key in a `map`'s subscript operator `[]` locates the value associated with that key in the `map`.

15.6.4 `map` Associative Container (Cont.)

- Insertions and deletions can be made *anywhere* in a `map`.
- If the order of the keys is not important, you can use `unordered_map` (header `<unordered_map>`) instead.
- Figure 15.18 demonstrates a `map` and uses the same features as Fig. 15.17 to demonstrate the subscript operator.
- Lines 27–28 use the subscript operator of class `map`.
- When the subscript is a key that is already in the `map` (line 27), the operator returns a reference to the associated value.

15.6.4 `map` Associative Container (Cont.)

- When the subscript is a key that is *not* in the `map` (line 18), the operator inserts the key in the `map` and returns a reference that can be used to associate a value with that key.
- Line 27 replaces the value for the key `25` (previously `33.333` as specified in line 16) with a new value, `9999.99`.
- Line 28 inserts a new key-value `pair` in the `map` (called creating an association).

```
1  // Fig. 15.18: fig15_18.cpp
2  // Standard Library class map class template.
3  #include <iostream>
4  #include <map> // map class-template definition
5  using namespace std;
6
7  int main()
8  {
9     map< int, double, less< int > > pairs;
10
11    // insert eight value_type objects in pairs
12    pairs.insert( make_pair( 15, 2.7 ) );
13    pairs.insert( make_pair( 30, 111.11 ) );
14    pairs.insert( make_pair( 5, 1010.1 ) );
15    pairs.insert( make_pair( 10, 22.22 ) );
16    pairs.insert( make_pair( 25, 33.333 ) );
17    pairs.insert( make_pair( 5, 77.54 ) ); // dup ignored
18    pairs.insert( make_pair( 20, 9.345 ) );
19    pairs.insert( make_pair( 15, 99.3 ) ); // dup ignored
20
21    cout << "pairs contains:\nKey\tValue\n";
22
```

Fig. 15.18 | Standard Library map class template. (Part 1 of 3.)

```
23      // walk through elements of pairs
24      for ( auto mapItem : pairs )
25         cout << mapItem.first << '\t' << mapItem.second << '\n';
26
27      pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
28      pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40
29
30      cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";
31
32      // use const_iterator to walk through elements of pairs
33      for ( auto mapItem : pairs )
34         cout << mapItem.first << '\t' << mapItem.second << '\n';
35
36      cout << endl;
37   } // end main
```

Fig. 15.18 | 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. 15.18 | Standard Library `map` class template. (Part 3 of 3.)

15.7 Container Adapters

- The three container adapters are `stack`, `queue` and `priority_queue`.
- Container 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.

15.7.1 `stack` Adapter

- Class stack (from header `<stack>`) enables insertions into and deletions from the underlying container at one end called the *top*, so a stack is commonly referred to as a *last-in*, *first-out* data structure.
- A `stack` can be implemented with `vector`, `list` or `deque`.
- This example creates three integer stacks, using `vector`, `list` and `deque` as the underlying data structure to represent the `stack`.
- By default, a `stack` is implemented with a `deque`.

15.7.1 `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).

15.7.1 `stack` Adapter (Cont.)

- Figure 15.19 demonstrates the `stack` adapter class.
- 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.

```
1  // Fig. 15.19: fig15_19.cpp
2  // Standard Library stack adapter class.
3  #include <iostream>
4  #include <stack> // stack adapter definition
5  #include <vector> // vector class-template definition
6  #include <list> // list class-template definition
7  using namespace std;
8
9  // pushElements function-template prototype
10 template< typename T > void pushElements( T &stackRef );
11
12 // popElements function-template prototype
13 template< typename T > void popElements( T &stackRef );
14
15 int main()
16 {
17    // stack with default underlying deque
18    stack< int > intDequeStack;
19
20    // stack with underlying vector
21    stack< int, vector< int > > intVectorStack;
22
23    // stack with underlying list
24    stack< int, list< int > > intListStack;
```

Fig. 15.19 | Standard Library `stack` adapter class. (Part 1 of 4.)

```
25
26      // push the values 0-9 onto each stack
27      cout << "Pushing onto intDequeStack: ";
28      pushElements( intDequeStack );
29      cout << "\nPushing onto intVectorStack: ";
30      pushElements( intVectorStack );
31      cout << "\nPushing onto intListStack: ";
32      pushElements( intListStack );
33      cout << endl << endl;
34
35      // display and remove elements from each stack
36      cout << "Popping from intDequeStack: ";
37      popElements( intDequeStack );
38      cout << "\nPopping from intVectorStack: ";
39      popElements( intVectorStack );
40      cout << "\nPopping from intListStack: ";
41      popElements( intListStack );
42      cout << endl;
43   } // end main
44
```

Fig. 15.19 | Standard Library `stack` adapter class. (Part 2 of 4.)

```
45  // push elements onto stack object to which stackRef refers
46  template< typename T > void pushElements( T &stackRef )
47  {
48     for ( int i = 0; i < 10; ++i )
49     {
50        stackRef.push( i ); // push element onto stack
51        cout << stackRef.top() << ' '; // view (and display) top element
52     } // end for
53  } // end function pushElements
54
55  // pop elements from stack object to which stackRef refers
56  template< typename T > void popElements( T &stackRef )
57  {
58     while ( !stackRef.empty() )
59     {
60        cout << stackRef.top() << ' '; // view (and display) top element
61        stackRef.pop(); // remove top element
62     } // end while
63  } // end function popElements
```

Fig. 15.19 | Standard Library `stack` adapter class. (Part 3 of 4.)

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

15.7.1 `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.

15.7.1 `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.

15.7.2 queue Adapter

- A queue is similar to a *waiting line*.
  - The item that has been in the queue the *longest* is the *next* one removed—so a queue is referred to as a first-in, first-out (FIFO) data structure.
- Class `queue` (from header `<queue>`) enables insertions at the *back* of the underlying data structure and deletions from the *front*.
- A `queue` can store its elements in objects of the Standard Library's `list` or `deque` containers.
- By default, a `queue` is implemented with a `deque`.

15.7.2 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).

15.7.2 queue Adapter (Cont.)

- Figure 15.20 demonstrates the `queue` adapter class.
- Line 9 instantiates a `queue` of `double`s.
- 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).

```
1  // Fig. 15.20: fig15_20.cpp
2  // Standard Library queue adapter class template.
3  #include <iostream>
4  #include <queue> // queue adapter definition
5  using namespace std;
6
7  int main()
8  {
9     queue< double > values; // queue with doubles
10
11    // push elements onto queue values
12    values.push( 3.2 );
13    values.push( 9.8 );
14    values.push( 5.4 );
15
16    cout << "Popping from values: ";
17
```

Fig. 15.20 | Standard Library queue adapter class templates.

15.7.2 queue Adapter (Cont.)

- 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*).

```
18      // pop elements from queue
19      while ( !values.empty() )
20      {
21         cout << values.front() << ' '; // view front element
22         values.pop(); // remove element
23      } // end while
24
25      cout << endl;
26   } // end main
```

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

Fig. 15.20 | Standard Library queue adapter class templates.

15.7.3 `priority_queue` Adapter (Cont.)

- Class `priority_queue` (from header `<queue>`) provides functionality that enables *insertions* in *sorted order* into the underlying data structure and deletions from the *front* of the underlying data structure.
- By default, a `priority_queue`'s elements are stored in a `vector`.
- 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`.

15.7.3 `priority_queue` Adapter (Cont.)

- This is usually accomplished by arranging the elements in a data structure called a heap (not to be confused with the heap for dynamically allocated memory) that always maintains the largest value (i.e., highest-priority element) at the front of the data structure.
- 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.
- Function `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, which then reorders the elements in priority order).

15.7.3 `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).

15.7.3 `priority_queue` Adapter (Cont.)

- Figure 15.21 demonstrates the `priority_queue` adapter class.
- Header `<queue>` must be included to use class `priority_queue`.

15.7.3 `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).

```
1  // Fig. 15.21: fig15_21.cpp
2  // Standard Library priority_queue adapter class.
3  #include <iostream>
4  #include <queue> // priority_queue adapter definition
5  using namespace std;
6
7  int main()
8  {
9     priority_queue< double > priorities; // create priority_queue
10
11    // push elements onto priorities
12    priorities.push( 3.2 );
13    priorities.push( 9.8 );
14    priorities.push( 5.4 );
15
16    cout << "Popping from priorities: ";
17
```

Fig. 15.21 | Standard Library `priority_queue` adapter class. (Part 1 of 2.)

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

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

Fig. 15.21 | Standard Library `priority_queue` adapter class. (Part 2 of 2.)