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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

15.2 Introduction to Containers

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

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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).

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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*.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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:

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

```
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.)

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.]

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

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`.

©1992-2014 by Pearson Education, Inc. All Rights Reserved.

Insert at specific location

```
#include <iostream>
#include <list>

int main() {
   std::list<int> myList {1, 2, 3, 4, 5};

   // insert 100 at index 2 (between 2 and 3)
   auto it = std::next(myList.begin(), 2);
   myList.insert(it, 100);

   // print the list
   for (int x : myList) {
      std::cout << x << " ";
   }
   std::cout << std::endl;

   return 0;
}
```

Output: 1 2 100 3 4 5