Standard Template Library (STL)

Lab 15: Vector and Map with Iterator

Rung-Bin Lin
International Bachelor Program in Informatics
Yuan Ze University

6/3/2022

21.1 Introduction to the Standard Template Library (STL)
- 21.1.1 Introduction to Containers
- 21.1.2 Introduction to Iterators
- 21.1.3 Introduction to Algorithms

21.2 Sequence Containers
- 21.2.1 `vector` Sequence Container
- 21.2.2 `list` Sequence Container
- 21.2.3 `deque` Sequence Container

21.3 Associative Containers
- 21.3.1 `multiset` Associative Container
- 21.3.2 `set` Associative Container
- 21.3.3 `multimap` Associative Container
- 21.3.4 `map` Associative Container

21.4 Container Adapters
- 21.4.1 `stack` Adapter
- 21.4.2 `queue` Adapter
- 21.4.3 `priority_queue` Adapter

21.5 Algorithms
21.5.1 `fill`, `fill_n`, `generate` and `generate_n`
21.5.2 `equal`, `mismatch` and `lexicographical_compare`
21.5.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if`
21.5.4 `replace`, `replace_if`, `replace_copy` and `replace_copy_if`
21.5.5 Mathematical Algorithms
21.5.6 Basic Searching and Sorting Algorithms
21.5.7 `swap`, `iter_swap` and `swap_ranges`
21.5.8 `copy_backward`, `merge`, `unique` and `reverse`
21.5.9 `inplace_merge`, `unique_copy` and `reverse_copy`
21.5.10 Set Operations
21.5.11 `lower_bound`, `upper_bound` and `equal_range`
21.5.12 Heapsort
21.5.13 `min` and `max`
21.5.14 STL Algorithms Not Covered in This Chapter
21.6 Class `bitset`
21.7 Function Objects
21.8 Wrap-Up
21.9 STL Web Resources

Introduction to the Standard Template Library (STL)

- STL's three key components
  - containers (popular templatized data structures)
  - Iterators
  - algorithms.
- The STL containers are data structures capable of storing objects of almost any data type (there are some restrictions).

Introduction to the Standard Template Library (STL) (Cont.)

- Each STL container has associated member functions.
  - A subset of these member functions is defined in all STL containers.
- STL iterators, which have properties similar to those of pointers, are used by programs to manipulate the STL-container elements.
  - In fact, standard arrays can be manipulated by STL algorithms, using standard pointers as iterators.
  - We'll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement.

Introduction to the Standard Template Library (STL) (Cont.)

- STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements (or entire containers).
- The STL provides approximately 70 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.

Introduction to Containers

- The STL container types are shown in Fig. 21.1.
- The containers are divided into three major categories—sequence containers, associative containers and container adapters.

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`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.
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. 21.1 | Standard Library container classes. (Part 2 of 2.)

Introduction to Containers (Cont.)

- The sequence containers represent linear data structures, such as vectors and linked lists.
- Associative containers are nonlinear containers that typically can locate elements stored in the containers quickly.
  - Such containers can store sets of values or key/value pairs.
- The sequence containers and associative containers are collectively referred to as the first-class containers.
- Stacks and queues actually are constrained versions of sequential containers.
- Type `string` (discussed in Chapter 18) supports the same functionality as a sequence container, but stores only character data.

Introduction to Containers (Cont.)

- Most STL containers provide similar functionality.
- Many generic operations, such as member function `size`, apply to all containers, and other operations apply to subsets of similar containers.

Member function	Description
default constructor	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
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.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.

Fig. 21.2 | Common member functions for most STL containers. (Part 1 of 3.)

Member function	Description
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or 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.
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the next position after the last element of the reversed container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 21.2 | Common member functions for most STL containers. (Part 3 of 3.)

Introduction to Containers (Cont.)

- Figure 21.3 shows the header files for STL containers
- Figure 21.4 shows the common `typedef`s (to create synonyms or aliases for lengthy type names) found in first-class containers.
- These `typedef`s are used in generic declarations of variables, parameters to functions and return values from functions.
- For example, `value_type` in each container is always a `typedef` that represents the type of value stored in the container.

Standard Library container header files

`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

Fig. 21.3 | Standard Library container header files.

typedef	Description
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference to the type of element stored in the container.
`const_reference`	A constant reference to the type of element stored in the container. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer to the type of element stored in the container.
`const_pointer`	A pointer to a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to the type of element stored in the container and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.
`const_reverse_iterator`	A constant reverse iterator that points to the type of element stored in the container and can be used only to *read* elements. This type of iterator is for iterating through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (operator `-` is not defined for iterators of `list`s and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

Fig. 21.4 | `typedef`s found in first-class containers. (Part 2 of 2.)

Introduction to Iterators

- Iterators have many features in common with pointers and are used to point to the elements of first-class containers (and for a few other purposes, as we'll see).
- Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container.
- Certain iterator operations are uniform across containers.
  - For example, the dereferencing operator (*) dereferences an iterator so that you can use the element to which it points.
  - The ++ operation on an iterator moves it to the next element of the container (much as incrementing a pointer into an array aims the pointer at the next element of the array).

Introduction to Iterators (Cont.)

- STL first-class containers provide member functions `begin` and `end`.
- Function `begin` returns an iterator pointing to the first element of the container.
- Function `end` returns an iterator pointing to the first element past the end of the container (an element that doesn't exist).

Introduction to Iterators (Cont.)

- If iterator `i` points to a particular element, then `++i` points to the "next" element and `*i` refers to the element pointed to by `i`.
- The iterator resulting from `end` is typically used in an equality or inequality comparison to determine whether the "moving iterator" (`i` in this case) has reached the end of the container.
- An object of type `iterator` refers to a container element that can be modified.
- An object of type `const_iterator` refers to a container element that cannot be modified.

Introduction to Iterators (Cont.)

- We use iterators with sequences (also called ranges).
- These sequences can be in containers, or they can be input sequences or output sequences.
- The program of Fig. 21.5 demonstrates input from the standard input (a sequence of data for input into a program), using an `istream_iterator`, and output to the standard output (a sequence of data for output from a program), using an `ostream_iterator`.
- The program inputs two integers from the user at the keyboard and displays the sum of the integers.

```
1  // Fig. 21.5: Fig21_05.cpp
2  // Demonstrating input and output with iterators.
3  #include <iostream>
4  #include <iterator> // ostream_iterator and istream_iterator
5  using namespace std;
6
7  int main()
8  {
9     cout << "Enter two integers: ";
10
11    // create istream_iterator for reading int values from cin
12    istream_iterator< int > inputInt( cin );
13
14    int number1 = *inputInt; // read int from standard input
15    ++inputInt; // move iterator to next input value
16    int number2 = *inputInt; // read int from standard input
17
18    // create ostream_iterator for writing int values to cout
19    ostream_iterator< int > outputInt( cout );
20
21    cout << "The sum is: ";
22    *outputInt = number1 + number2; // output result to cout
23    cout << endl;
24 } // end main
```

```
Enter two integers: 12 25
The sum is: 37
```

Fig. 21.5 | Input and output stream iterators. (Part 2 of 2.)

Introduction to Iterators (Cont.)

- Figure 21.6 shows the categories of STL iterators.
- Each category provides a specific set of functionality.
- Figure 21.7 illustrates the hierarchy of iterator categories.
- As you follow the hierarchy from top to bottom, each iterator category supports all the functionality of the categories above it in the figure.
- Thus the “weakest” iterator types are at the top and the most powerful one is at the bottom.
- Note that this is not an inheritance hierarchy.

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Fig. 21.6 | Iterator categories.

Introduction to Iterators (Cont.)

- The iterator category that each container supports determines whether that container can be used with specific algorithms in the STL.
- Containers that support random-access iterators can be used with all algorithms in the STL.
- As we'll see, pointers into arrays can be used in place of iterators in most STL algorithms, including those that require random-access iterators.
- Figure 21.8 shows the iterator category of each of the STL containers.
- The first-class containers (`vector`s, `deque`s, `list`s, `set`s, `multiset`s, `map`s and `multimap`s), `string`s and arrays are all traversable with iterators.

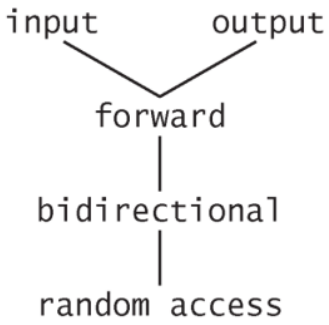

Fig. 21.7 | Iterator category hierarchy.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Fig. 21.8 | Iterator types supported by each container.

Introduction to Iterators (Cont.)

- Figure 21.9 shows the predefined iterator `typedef`s that are found in the class definitions of the STL containers.
- Not every `typedef` is defined for every container.
- We use `const` versions of the iterators for traversing read-only containers.
- We use reverse iterators to traverse containers in the reverse direction.

Predefined `typedef`s for iterator types	Direction of ++	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

Fig. 21.9 | Iterator `typedef`s.

Introduction to Iterators (Cont.)

- Figure 21.10 shows some operations that can be performed on each iterator type.
- The operations for each iterator type include all operations preceding that type in the figure.

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

Fig. 21.10 | Iterator operations for each type of iterator. (Part 1 of 3.)

Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Fig. 21.10 | Iterator operations for each type of iterator. (Part 3 of 3.)

Introduction to Algorithms

- STL algorithms can be used generically across a variety of containers.
- STL provides many algorithms you'll use frequently to manipulate containers.
- Inserting, deleting, searching, sorting and others are appropriate for some or all of the STL containers.
- The STL includes approximately 70 standard algorithms.
- The algorithms operate on container elements only indirectly through iterators.
- Many algorithms operate on sequences of elements defined by pairs of iterators—one pointing to the first element of the sequence and one pointing to one element past the last element.

Introduction to Algorithms (Cont.)

- Algorithms often return iterators that indicate the results of the algorithms. For example,
  - Algorithm `find`, for example, locates an element and returns an iterator to that element.
  - If the element is not found, `find` returns the "one past the `end`" iterator that was passed in to define the end of the range to be searched, which can be tested to determine whether an element was not found.
  - The `find` algorithm can be used with any first-class STL container.
- Some algorithms demand powerful iterators; e.g., `sort` demands random-access iterators.

Introduction to Algorithms (Cont.)

- Figure 21.11 shows many of the mutating-sequence algorithms—i.e., the algorithms that result in modifications of the containers to which the algorithms are applied.

Mutating-sequence algorithms			
copy	partition	replace_copy	stable_partition
copy_backward	random_shuffle	replace_copy_if	swap
fill	remove	replace_if	swap_ranges
fill_n	remove_copy	reverse	transform
generate	remove_copy_if	reverse_copy	unique
generate_n	remove_if	rotate	unique_copy
iter_swap	replace	rotate_copy	Sort

Fig. 21.11 | Mutating-sequence algorithms.

https://www.cplusplus.com/reference/algorithm/

Introduction to Algorithms (Cont.)

- Figure 21.12 shows many of the nonmodifying sequence algorithms—i.e., the algorithms that do not result in modifications of the containers to which they're applied.
- Figure 21.13 shows the numerical algorithms of the header file `<numeric>`.

Nonmodifying sequence algorithms			
adjacent_find	equal	find_end	mismatch
count	find	find_first_of	search
count_if	find_each	find_if	search_n

Fig. 21.12 | Nonmodifying sequence algorithms.

Numerical algorithms from header file <numeric>	
accumulate	partial_sum
inner_product	adjacent_difference

Fig. 21.13 | Numerical algorithms from header file <numeric>.

Sequence Containers

- The C++ Standard Template Library provides three sequence containers—`vector`, `list` and `deque`.
- Class template `vector` and class template `deque` both are based on arrays.
- Class template `list` implements a linked-list data structure similar to our `List` class presented in Chapter 20, but more robust.
- In addition to the common operations described in Fig. 21.2, the sequence containers have several other common operations
  - `front` to return a reference to the first element in a non-empty container
  - `back` to return a reference to the last element in a non-empty container
  - `push_back` to insert a new element at the end of the container
  - `pop_back` to remove the last element of the container.

vector Sequence Container (Cont.)

- Class template `vector` provides a data structure with contiguous memory locations.
- This enables efficient, direct access to any element of a vector via the subscript operator `[]`, exactly as with a C or C++ “raw” array.
- Class template `vector` is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted.
- When a `vector`’s memory is exhausted, the `vector` allocates a larger contiguous area of memory, copies the original elements into the new memory and deallocates the old memory.

vector Sequence Container (Cont.)

- An important part of every container is the type of iterator it supports.
- This determines which algorithms can be applied to the container.
- A `vector` supports random-access iterators—i.e., all iterator operations shown in Fig. 21.10 can be applied to a `vector` iterator.
- All STL algorithms can operate on a `vector`.
- The iterators for a `vector` are sometimes implemented as pointers to elements of the `vector`.

vector Sequence Container (Cont.)

- Figure 21.14 illustrates several functions of the `vector` class template.
- Many of these functions are available in every first-class container.
- You must include header file `<vector>` to use class template `vector`.

```
1  // Fig. 21.14: Fig21_14.cpp
2  // Demonstrating Standard Library vector class template.
3  #include <iostream>
4  #include <vector> // vector class-template definition
5  using namespace std;
6
7  // prototype for function template printVector
8  template < typename T > void printVector( const vector< T > &integers2 );
9
10 int main()
11 {
12    const int SIZE = 6; // define array size
13    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
14    vector< int > integers; // create vector of ints
15
16    cout << "The initial size of integers is: " << integers.size()
17       << "\nThe initial capacity of integers is: " << integers.capacity();
18
19    // function push_back is in every sequence collection
20    integers.push_back( 2 );
21    integers.push_back( 3 );
22    integers.push_back( 4 );
23
```

Fig. 21.14 | Standard Library `vector` class template. (Part 1 of 3.)

```
24       cout << "\nThe size of integers is: " << integers.size()
25          << "\nThe capacity of integers is: " << integers.capacity();
26       cout << "\n\nOutput array using pointer notation: ";
27
28       // display array using pointer notation
29       for ( int *ptr = array; ptr != array + SIZE; ptr++ )
30          cout << *ptr << ' ';
31
32       cout << "\nOutput vector using iterator notation: ";
33       printVector( integers );
34       cout << "\nReversed contents of vector integers: ";
35
36       // two const reverse iterators
37       vector< int >::const_reverse_iterator reverseIterator;
38       vector< int >::const_reverse_iterator tempIterator = integers.rend();
39
40       // display vector in reverse order using reverse_iterator
41       for ( reverseIterator = integers.rbegin();
42          reverseIterator!= tempIterator; ++reverseIterator )
43          cout << *reverseIterator << ' ';
44
45       cout << endl;
46    } // end main
47
```

Fig. 21.14 | Standard Library `vector` class template. (Part 2 of 3.)

```
48  // function template for outputting vector elements
49  template < typename T > void printVector( const vector< T > &integers2 )
50  {
51     typename vector< T >::const_iterator constIterator; // const_iterator
52
53     // display vector elements using const_iterator
54     for ( constIterator = integers2.begin();
55        constIterator != integers2.end(); ++constIterator )
56        cout << *constIterator << ' ';
57  } // end function printVector
```

```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Fig. 21.14 | Standard Library `vector` class template. (Part 3 of 3.)

vector Sequence Container (Cont.)

- Figure 21.15 illustrates functions that enable retrieval and manipulation of the elements of a `vector`.
- Line 15 uses an overloaded `vector` constructor that takes two iterators as arguments to initialize `integers`.

```
1  // Fig. 21.15: Fig21_15.cpp
2  // Testing Standard Library vector class template
3  // element-manipulation functions.
4  #include <iostream>
5  #include <vector> // vector class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator iterator
8  #include <stdexcept> // out_of_range exception
9  using namespace std;
10
11 int main()
12 {
13    const int SIZE = 6;
14    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15    vector< int > integers( array, array + SIZE );
16    ostream_iterator< int > output( cout, " " );
17
18    cout << "Vector integers contains: ";
19    copy( integers.begin(), integers.end(), output );
20
21    cout << "\nFirst element of integers: " << integers.front()
22       << "\nLast element of integers: " << integers.back();
23
```

Fig. 21.15 | `vector` class template element-manipulation functions. (Part 1 of 4.)

```
24       integers[ 0 ] = 7; // set first element to 7
25       integers.at( 2 ) = 10; // set element at position 2 to 10
26
27       // insert 22 as 2nd element
28       integers.insert( integers.begin() + 1, 22 );
29
30       cout << "\n\nContents of vector integers after changes: ";
31       copy( integers.begin(), integers.end(), output );
32
33       // access out-of-range element
34       try
35       {
36          integers.at( 100 ) = 777;
37       } // end try
38       catch ( out_of_range &outOfRange ) // out_of_range exception
39       {
40          cout << "\n\nException: " << outOfRange.what();
41       } // end catch
42
43       // erase first element
44       integers.erase( integers.begin() );
45       cout << "\n\nVector integers after erasing first element: ";
46       copy( integers.begin(), integers.end(), output );
47
```

Fig. 21.15 | `vector` class template element-manipulation functions. (Part 2 of 4.)

```
48      // erase remaining elements
49      integers.erase( integers.begin(), integers.end() );
50      cout << "\nAfter erasing all elements, vector integers "
51         << ( integers.empty() ? "is" : "is not" ) << " empty";
52
53      // insert elements from array
54      integers.insert( integers.begin(), array, array + SIZE );
55      cout << "\n\nContents of vector integers before clear: ";
56      copy( integers.begin(), integers.end(), output );
57
58      // empty integers; clear calls erase to empty a collection
59      integers.clear();
60      cout << "\nAfter clear, vector integers "
61         << ( integers.empty() ? "is" : "is not" ) << " empty" << endl;
62   } // end main
```

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

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

Exception: invalid vector<T> subscript

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

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

Fig. 21.15 | `vector` class template element-manipulation functions. (Part 4 of 4.)

Sorting data in vector using STL Algorithm

```
1  // sort algorithm example
2  #include <iostream>      // std::cout
3  #include <algorithm>     // std::sort
4  #include <vector>        // std::vector
5
6  bool myfunction (int i,int j) { return (i<j); }
7
8  struct myclass {
9    bool operator() (int i,int j) { return (i<j);}
10 } myobject;
11
12 int main () {
13   int myints[] = {32,71,12,45,26,80,53,33};
14   std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33
15
16   // using default comparison (operator <):
17   std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33
18
19   // using function as comp
20   std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)
21
22   // using object as comp
23   std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)
24
25   // print out content:
26   std::cout << "myvector contains:";
27   for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
28     std::cout << ' ' << *it;
29   std::cout << '\n';
30
31   return 0;
32 }
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

Associative Containers

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

Associative Containers (Cont.)

- Classes `multimap` and `map` provide operations for manipulating values associated with keys (these values are sometimes referred to as mapped values).
- The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.
- In addition to the common member functions of all containers presented in Fig. 21.2, all associative containers also support several other member functions, including `find`, `lower_bound`, `upper_bound` and `count`.
- Examples of each of the associative containers and the common associative container member functions are presented in the next several subsections.

multiset Associative Container (Cont.)

- The `multiset` associative container provides fast storage and retrieval of keys and allows duplicate keys.
- The ordering of the elements is determined by a comparator function object.
- For example, in an integer `multiset`, elements can be sorted in ascending order by ordering the keys with comparator function object `less<int>`.
- We discuss function objects in detail in Section 21.7.
- The data type of the keys in all associative containers must support comparison properly based on the comparator function object specified—keys sorted with `less<T>` must support comparison with `operator<`.

```
1  // Fig. 21.19: Fig21_19.cpp
2  // Testing Standard Library class multiset
3  #include <iostream>
4  #include <set> // multiset class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  // define short name for multiset type used in this program
10 typedef multiset< int, less< int > > Ims;
11
12 int main()
13 {
14    const int SIZE = 10;
15    int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
16    Ims intMultiset; // Ims is typedef for "integer multiset"
17    ostream_iterator< int > output( cout, " " );
18
19    cout << "There are currently " << intMultiset.count( 15 )
20       << " values of 15 in the multiset\n";
21
```

Fig. 21.19 | Standard Library `multiset` class template. (Part 1 of 4.)

```
22      intMultiset.insert( 15 ); // insert 15 in intMultiset
23      intMultiset.insert( 15 ); // insert 15 in intMultiset
24      cout << "After inserts, there are " << intMultiset.count( 15 )
25         << " values of 15 in the multiset\n\n";
26
27      // iterator that cannot be used to change element values
28      Ims::const_iterator result;
29
30      // find 15 in intMultiset; find returns iterator
31      result = intMultiset.find( 15 );
32
33      if ( result != intMultiset.end() ) // if iterator not at end
34         cout << "Found value 15\n"; // found search value 15
35
36      // find 20 in intMultiset; find returns iterator
37      result = intMultiset.find( 20 );
38
39      if ( result == intMultiset.end() ) // will be true hence
40         cout << "Did not find value 20\n"; // did not find 20
41
42      // insert elements of array a into intMultiset
43      intMultiset.insert( a, a + SIZE );
44      cout << "\nAfter insert, intMultiset contains:\n";
45      copy( intMultiset.begin(), intMultiset.end(), output );
```

Fig. 21.19 | Standard Library `multiset` class template. (Part 2 of 4.)

```
46
47      // determine lower and upper bound of 22 in intMultiset
48      cout << "\n\nLower bound of 22: "
49         << *( intMultiset.lower_bound( 22 ) );
50      cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );
51
52      // p represents pair of const_iterators
53      pair< Ims::const_iterator, Ims::const_iterator > p;
54
55      // use equal_range to determine lower and upper bound
56      // of 22 in intMultiset
57      p = intMultiset.equal_range( 22 );
58
59      cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
60         << *( p.first ) << "\n   Upper bound: " << *( p.second );
61      cout << endl;
62   } // 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. 21.19 | Standard Library `multiset` class template. (Part 4 of 4.)

set Associative Container

- The `set` associative container is used for fast storage and retrieval of unique keys.
- The implementation of a `set` is identical to that of a `multiset`, except that a `set` must have unique keys.
- Therefore, if an attempt is made to insert a duplicate key into a `set`, the duplicate is ignored; because this is the intended mathematical behavior of a set, we do not identify it as a common programming error.
- A `set` supports bidirectional iterators (but not random-access iterators).
- Figure 21.20 demonstrates a `set` of `doubles`.
- Header file `<set>` must be included to use class `set`.

```
1  // Fig. 21.20: Fig21_20.cpp
2  // Standard Library class set test program.
3  #include <iostream>
4  #include <set>
5  #include <algorithm>
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  // define short name for set type used in this program
10 typedef set< double, less< double > > DoubleSet;
11
12 int main()
13 {
14    const int SIZE = 5;
15    double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
16    DoubleSet doubleSet( a, a + SIZE );
17    ostream_iterator< double > output( cout, " " );
18
19    cout << "doubleSet contains: ";
20    copy( doubleSet.begin(), doubleSet.end(), output );
21
22    // p represents pair containing const_iterator and bool
23    pair< DoubleSet::const_iterator, bool > p;
```

Fig. 21.20 | Standard Library `set` class template. (Part 1 of 3.)

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

```
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. 21.20 | Standard Library `set` class template. (Part 3 of 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 >`.

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 21.21 demonstrates the `multimap` associative container.
- Header file `<map>` must be included to use class `multimap`.

```
1  // Fig. 21.21: Fig21_21.cpp
2  // Standard Library class multimap test program.
3  #include <iostream>
4  #include <map> // multimap class-template definition
5  using namespace std;
6
7  // define short name for multimap type used in this program
8  typedef multimap< int, double, less< int > > Mmid;
9
10 int main()
11 {
12    Mmid pairs; // declare the multimap pairs
13
14    cout << "There are currently " << pairs.count( 15
15       << " pairs with key 15 in the multimap\n";
16
17    // insert two value_type objects in pairs
18    pairs.insert( Mmid::value_type( 15, 2.7 ) );
19    pairs.insert( Mmid::value_type( 15, 99.3 ) );
20
21    cout << "After inserts, there are " << pairs.count( 15 )
22       << " pairs with key 15\n\n";
23
```

Fig. 21.21 | Standard Library `multimap` class template. (Part 1 of 3.)

```
24      // insert five value_type objects in pairs
25      pairs.insert( Mmid::value_type( 30, 111.11 ) );
26      pairs.insert( Mmid::value_type( 10, 22.22 ) );
27      pairs.insert( Mmid::value_type( 25, 33.333 ) );
28      pairs.insert( Mmid::value_type( 20, 9.345 ) );
29      pairs.insert( Mmid::value_type( 5, 77.54 ) );
30
31      cout << "Multimap pairs contains:\nKey\tValue\n";
32
33      // use const_iterator to walk through elements of pairs
34      for ( Mmid::const_iterator iter = pairs.begin();
35         iter != pairs.end(); ++iter )
36         cout << iter->first << '\t' << iter->second << '\n';
37
38      cout << endl;
39   } // end main
```

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

map Associative Container

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

```
1  // Fig. 21.22: Fig21_22.cpp
2  // Standard Library class map test program.
3  #include <iostream>
4  #include <map> // map class-template definition
5  using namespace std;
6
7  // define short name for map type used in this program
8  typedef map< int, double, less< int > > Mid;
9
10 int main()
11 {
12    Mid pairs;
13
14    // insert eight value_type objects in pairs
15    pairs.insert( Mid::value_type( 15, 2.7 ) );
16    pairs.insert( Mid::value_type( 30, 111.11 ) );
17    pairs.insert( Mid::value_type( 5, 1010.1 ) );
18    pairs.insert( Mid::value_type( 10, 22.22 ) );
19    pairs.insert( Mid::value_type( 25, 33.333 ) );
20    pairs.insert( Mid::value_type( 5, 77.54 ) ); // dup ignored
21    pairs.insert( Mid::value_type( 20, 9.345 ) );
22    pairs.insert( Mid::value_type( 15, 99.3 ) ); // dup ignored
23
```

Fig. 21.22 | Standard Library `map` class template. (Part 1 of 3.)

```
24      cout << "pairs contains:\nKey\tValue\n";
25
26      // use const_iterator to walk through elements of pairs
27      for ( Mid::const_iterator iter = pairs.begin();
28         iter != pairs.end(); ++iter )
29         cout << iter->first << '\t' << iter->second << '\n';
30
31      pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
32      pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40
33
34      cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";
35
36      // use const_iterator to walk through elements of pairs
37      for ( Mid::const_iterator iter2 = pairs.begin();
38         iter2 != pairs.end(); ++iter2 )
39         cout << iter2->first << '\t' << iter2->second << '\n';
40
41      cout << endl;
42   } // end main
```

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

Lab 15: Vector and Map with Iterator

- Given two files, x.lef and b17.def, x.lef consists of some macro's and b17.def consists of references to the macro's in x.lef.
  - Each macro in x.lef starts with a keyword MACRO, followed by macro's name. Each macro has a size specified by its width and height. The area of a macro is equal to the product of its width and height. An example of macro's description is shown in page 61.
  - A reference (or an instance) in b17.def to a macro in x.lef is specified by a statement as follows:

    - U1648 NOR2_xp33_75t + PLACED (674496 277920) FN ;

    It starts with a -, followed by instance's name (U1648), and then followed by macro's name (NOR2_xp33_75t).

- The first problem is to calculate the total area of all the macro instances found in COMPONENTS section in b17.def. For example, if a macro is referred twice (i.e., two instances of the macro) in b17.def, then the total area contributed by this macro is twice the area of the macro. While calculating the total area, you should not count the area of the following macros: FILLER_75t, FILLER_xp5_75t, TAPCELL_WITH_FILLER_75t, TAPCELL_75t, DECAP_x10_75t, DECAP_x1_75t, DECAP_x2_75t, DECAP_x4_75t, DECAP_x6_75t, TIEHI_x1_75t, TIELO_x1_75t.
  - Here, you should use **map** to store all the macro's found in X.LEF. The key is the macro's name. The value may include macro's name, macro's area, and count (the number of times being used in b17.def). Then, the name of an instance found in b17.def should be used as a key to find out the macro's area for the instance. Hence, I suggest you should create a template (class) for this purpose. So the value associated with a key becomes an object of the above class. To get access to the key and the value, you can refer to the code given in Fig. 21.21.
  - You should use **set** to avoid adding the area of an instance of the above macros into the total area. That is, using **set** to check whether an instance is one of the above macros.

- The second problem is to find out the top 10 frequently used macros, but not including the following macros: FILLER_75t, FILLER_xp5_75t, TAPCELL_WITH_FILLER_75t,TAPCELL_75t, DECAP_x10_75t, DECAP_x1_75t, DECAP_x2_75t, DECAP_x4_75t, DECAP_x6_75t, TIEHI_x1_75t, TIELO_x1_75t;
- The third problem is to store all the instance names found in b17.def in a vector and use the STL sort algorithm to sort the names in the ascending lexicographic order. Then, print out the instance names at locations 100, 500, 1000, 8000, 15000, and 20000 respectively on a line.
- **Requirements**
  - **The runtime of your program should be less than 3 seconds.**
    - **Add #include<ctime> into the program**
    - **After opening the two files in the main() function, add the statement time_t startTime = time(NULL);**
    - **Just before the return 0; in the main() function, add the following two statements.**

      ```
      time_t stopTime = time(NULL);
      cout << "\n  Runtime: " << stopTime - startTime << endl;
      ```
  - Iterators for vectors and maps should be used in the program.

MACRO in LEF File

```
MACRO INV_x1_75t
  CLASS CORE ;
  ORIGIN 0 0 ;
  FOREIGN INV_x1_75t 0 0 ;
  SIZE 0.648 BY 1.08 ;
  SYMMETRY X Y ;
  SITE coreSite ;
  PIN A
    DIRECTION INPUT ;
    USE SIGNAL ;
    PORT
      LAYER M1 ;
        RECT 0.072 0.504 0.312 0.576 ;
        RECT 0.072 0.9 0.22 0.972 ;
        RECT 0.072 0.108 0.22 0.18 ;
        RECT 0.072 0.108 0.144 0.972 ;
    END
  END A
  PIN VDD
    DIRECTION INOUT ;
    USE POWER ;
    SHAPE ABUTMENT ;
    PORT
      LAYER M1 ;
        RECT 0 1.044 0.648 1.116 ;
    END
  END VDD
  PIN VSS
    DIRECTION INOUT ;
    USE GROUND ;
    SHAPE ABUTMENT ;
    PORT
      LAYER M1 ;
        RECT 0 -0.036 0.648 0.036 ;
    END
  END VSS
  PIN Y
    DIRECTION INPUT ;
    USE SIGNAL ;
    PORT
      LAYER M1 ;
        RECT 0.376 0.9 0.576 0.972 ;
        RECT 0.504 0.108 0.576 0.972 ;
        RECT 0.376 0.108 0.576 0.18 ;
    END
  END Y
END INV_x1_75t
```

COMPONENTS Section in b17.DEF

```
COMPONENTS 27327 ;
- FE_OFC71_FE_OFN37_reset BUF_x2_75t + SOURCE TIMING + PLACED ( 576000 576000 )
FS
 ;
- FE_OCPC91_n1161 BUF_x1_75t + SOURCE TIMING + PLACED ( 612288 92160 ) FS
 ;
- FE_OCPC90_n1027 BUF_xp67_75t + SOURCE TIMING + PLACED ( 569088 139680 ) FN
 ;
- FE_OCPC88_n1143 BUF_xp33_75t + SOURCE TIMING + PLACED ( 715968 74880 ) FS
 ;
- FE_OCPC86_n2057 BUF_xp33_75t + SOURCE TIMING + PLACED ( 358272 260640 ) N
 ;
- FE_OCPC84_n1040 BUF_x2_75t + SOURCE TIMING + PLACED ( 591552 144000 ) FS
 ;
- FE_OCPC82_n329 BUF_xp33_75t + SOURCE TIMING + PLACED ( 467136 239040 ) S
 ;
- FE_RC_26_0 OR2_x1_75t + SOURCE TIMING + PLACED ( 207936 398880 ) FN
 ;
- FE_RC_25_0 NOR2_x1_75t + SOURCE TIMING + PLACED ( 228672 385920 ) FS
 ;
....
END COMPONENTS
```

Output

```
LEF file name: x.lef
DEF file name: b17.def

Number of cell instances: 27327   Number of macros: 217
Total Area: 25001.1

The top 10 frequently used macros:
       INV_xp33_75t  2494
     NAND2_xp33_75t  2009
      NOR2_xp33_75t  1624
     OAI21_xp33_75t  1310
 ASYNC_DFFH_x1_75t  1309
     AOI21_xp33_75t  900
     AOI22_xp33_75t  798
     OAI22_xp33_75t  655
        AND2_x1_75t  634
       XOR2_xp5_75t  592

Instances at locations 100, 500, 1000, 8000, 15000, and 20000 respectively:
   100: FE_OFC12_datao_31
   500: FILLER_1028
  1000: FILLER_1240
  8000: FILLER_7541
 15000: P2\/U4221
 20000: P3\/U4220

Runtime (in seconds): 0
```

Should be not larger than 3.