COURSE TECHNOLOGY
CENGAGE Learning

Data Structures Using C++ 2E

Chapter 4

Standard Template Library (STL) I

Objectives

- Learn about the Standard Template Library (STL)
- Become familiar with the three basic components of the STL: containers, iterators, and algorithms
- Explore how `vector` and `deque` containers are used to manipulate data in a program
- Discover the use of iterators

Components of the STL

- Program's main objective is to manipulate data and generate results
  - Requires ability to store data, access data, and manipulate data
- STL components
  - Containers: manage objects of a given type
  - Iterators: step through container elements
  - Algorithms: manipulate data
- Containers and iterators
  - Class templates

Container Types

- STL containers categories
  - Sequence containers (sequential containers)
    - vector, deque, list, array, etc.
  - Associative containers
    - set, multiset, map, multimap, etc.
  - Container adapters
    - stack: LIFO, (default) adapts deque
    - queue: FIFO, (default) adapts deque
    - priority_queue: items in sorted order, (default) adapts vector

Sequence Containers

- Every object has a specific position
- Predefined sequence containers
  - `vector`, `deque`, `list`
- Sequence container `vector`
  - Logically: same as arrays
  - Processed like arrays
- All containers
  - Use same names for common operations
  - Have container specific operations

Sequence Container: `vector`

- Vector container
  - Stores, manages objects of a type in a dynamic array
  - Elements accessed randomly
  - Time-consuming item insertion: middle, beginning
  - Fast item insertion: end
  - Class name: `vector`
- Header file containing the `class vector`

```
#include <vector>
```

- Examples

```
vector<int> intList;
vector<string> stringList;
```

Sequence Container: `vector` (cont'd.)

- Declaring vector objects

TABLE 4-1 Various ways to declare and initialize a vector container

Statement	Effect
`vector<elementType> vecList;`	Creates an empty vector, `vecList`, without any elements. (The default constructor is invoked.)
`vector<elementType> vecList(otherVecList);`	Creates a vector, `vecList`, and initializes `vecList` to the elements of the vector `otherVecList`. `vecList` and `otherVecList` are of the same type.
`vector<elementType> vecList(size);`	Creates a vector, `vecList`, of size `size`. `vecList` is initialized using the default constructor.
`vector<elementType> vecList(n, elem);`	Creates a vector, `vecList`, of size `n`. `vecList` is initialized using `n` copies of the element `elem`.
`vector<elementType> vecList(begin, end);`	Creates a vector, `vecList`. `vecList` is initialized to the elements in the range `[begin, end)`, that is, all elements in the range `begin...end-1`.

Sequence Container: `vector` (cont'd.)

EXAMPLE 4-1

a. The following statement declares **intList** to be an empty vector container and the element type is **int**.

```
vector<int> intList;
```

b. The following statement declares **intList** to be a vector container of size 10 and the element type is **int**. The elements of **intList** are initialized to **0**.

```
vector<int> intList(10);
```

c. The following statement declares **intList** to be a vector container of size 5 and the element type is **int**. The container **intList** is initialized using the elements of the array.

```
int intArray[5] = {2,4,6,8,10};
vector<int> intList(intArray, intArray + 5);
```

The container **intList** is initialized using the elements of the array **intArray**. That is, **intList = {2,4,6,8,10}**.

Sequence Container: `vector` (cont'd.)

- Manipulating data stored in a vector sequence container
  - Item insertion, deletion
  - Stepping through the elements of a vector array

TABLE 4-2 Operations to access the elements of a vector container

Expression	Effect
`vecList.at(index)`	Returns the element at the position specified by **`index`**.
`vecList[index]`	Returns the element at the position specified by **`index`**.
`vecList.front()`	Returns the first element. (Does not check whether the container is empty.)
`vecList.back()`	Returns the last element. (Does not check whether the container is empty.)

Sequence Container: `vector` (cont'd.)

- The elements in a vector can be processed just as they can in an array

EXAMPLE 4-2

Consider the following statement, which declares **intList** to be a vector container of size 5 and the element type is **int**.

```
vector<int> intList(5);
```

You can use a loop, such as the following, to store elements into **intList**:

```
for (int j = 0; j < 5; j++)
    intList[j] = j;
```

Similarly, you can use a **for** loop to output the elements of **intList**.

Sequence Container: `vector` (cont'd.)

TABLE 4-3 Various operations on a vector container

Expression	Effect
`vecList.clear()`	Deletes all elements from the container.
`vecList.erase(position)`	Deletes the element at the position specified by position.
`vecList.erase(beg, end)`	Deletes all elements starting at `beg` until `end-1`.
`vecList.insert(position, elem)`	A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned.
`vecList.insert(position, n, elem)`	`n` copies of `elem` are inserted at the position specified by `position`.
`vecList.insert(position, beg, end)`	A copy of the elements, starting at `beg` until `end-1`, is inserted into `vecList` at the position specified by `position`.

Sequence Container: `vector` (cont'd.)

TABLE 4-3 Various operations on a vector container (cont'd.)

Expression	Effect
`vecList.push_back(elem)`	A copy of `elem` is inserted into `vecList` at the end.
`vecList.pop_back()`	Deletes the last element.
`vecList.resize(num)`	Changes the number of elements to `num`. If `size()`, that is, the number of elements in the container increases, the default constructor creates the new elements.
`vecList.resize(num, elem)`	Changes the number of elements to `num`. If `size()` increases, the default constructor creates the new elements.

Sequence Container: `vector` (cont'd.)

- Function `push_back`
  - Adds element to end of container
  - You cannot use array subscripting operator `[]` to add an element beyond it's size unless you grow the size of the container first
  - Container size grows automatically as needed with `push_back`
  - Used when declaring vector container
    - Specific size unknown

Sequence Container: `vector` (cont'd.)

EXAMPLE 4-3

The following statement declares **intList** to be a vector container of size 0.

```
vector<int> intList;
```

To add elements into **intList**, we can use the function **push_back** as follows:

```
intList.push_back(34);
intList.push_back(55);
```

After these statements execute, the size of **intList** is 2 and **intList = {34, 55}**. Of course, you could have used the **resize** function to increase the size of **intList** and then use the array subscripting operator. However, at times, the **push_back** function is more convenient because it does not need to know the size of the container; it simply adds elements at the end.

Sequence Container: `vector` (cont'd.)

- `class vector`
  - Provides various operations to process vector container elements
  - Iterator
    - Argument position in STL terminology
    - Works just like a pointer
    - Used to step through the elements of a container

Declaring an Iterator to a Vector Container

- You can process vector container like an array
  - Using array subscripting operator `[]`
- Or process vector container elements
  - Using an iterator
- `class vector`: function `insert`
  - `vecList::insert(position, elem)`
  - Insert element at a specific vector container position
  - Uses an iterator `position`
- `class vector`: function `erase`
  - `vecList::erase(position)`
  - Remove element
  - Uses an iterator `position`

Declaring an Iterator to a Vector Container (cont'd.)

- `class vector` contains `typedef iterator`
  - Declared as a public member
  - Must use the container name (vector), element type, and scope resolution operator to use typedef iterator
  - Vector container iterator
    - Declared using `typedef iterator`
  - Example

    ```
    vector<int>::iterator intVecIter;
    ```

    declares `intVecIter` to be an iterator into a vector container of type `int`

Iterator definition in STL vector

```
template <class _Tp>
class vector : protected _Vector_base<_Tp>
{
public:
  typedef _Tp value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type* iterator;
  …
};
```

Declaring an Iterator to a Vector Container (cont'd.)

- Requirements for using `typedef iterator`
  - Container name (vector)
  - Container element type
  - Scope resolution operator
- `++intVecIter`
  - Advances iterator `intVecIter` to next element into the container
- `*intVecIter`
  - Returns element at current iterator position

Declaring an Iterator to a Vector Container (cont'd.)

- Using an iterator into a vector container
  - Manipulating element type to be `int`

```
vector<int> intList;                    //Line 1
vector<int>::iterator intVecIter;       //Line 2
```

Containers and the Functions `begin` and `end`

- `begin()`
  - Returns position of the first element into the container
- `end()`
  - Returns position one after the last element of the container

After the following statement executes:

```
intVecIter = intList.begin();
```

the iterator **intVecIter** points to the first element into the container **intList**.

The following **for** loop uses an iterator to output the elements of **intList** onto the standard output device:

```
for (intVecIter = intList.begin(); intVecIter != intList.end();
                                   ++intVecList)
    cout << *intVecIter << " ";
```

Containers and the Functions `begin` and `end`

- What is the value of vecList after the following code is executed?

EXAMPLE 4-4

Consider the following statements:

```
int intArray[7] = {1, 3, 5, 7, 9, 11, 13};     //Line 1
vector<int> vecList(intArray, intArray + 7};   //Line 2
vector<int>::iterator  intVecIter;             //Line 3
```

The statement in Line 2 declares and initializes the vector container **vecList**. Now consider the following statements:

```
intVecIter = vecList.begin();      //Line 4
++intVecIter;                      //Line 5
vecList.insert(intVecIter, 22};    //Line 6
```

Containers and the Functions `begin` and `end` (cont'd.)

TABLE 4-4 Functions to determine the size of a vector container

Expression	Effect
`vecCont.capacity()`	Returns the maximum number of elements that can be inserted into the container `vecCont` without reallocation.
`vecCont.empty()`	Returns **`true`** if the container `vecCont` is empty and **`false`** otherwise.
`vecCont.size()`	Returns the number of elements currently in the container `vecCont`.
`vecCont.max_size()`	Returns the maximum number of elements that can be inserted into the container `vecCont`.

Example: vector

Example

```
1  // comparing size, capacity and max_size
2  #include <iostream>
3  #include <vector>
4
5  int main ()
6  {
7    std::vector<int> myvector;
8
9    // set some content in the vector:
10   for (int i=0; i<100; i++) myvector.push_back(i);
11
12   std::cout << "size: " << myvector.size() << "\n";
13   std::cout << "capacity: " << myvector.capacity() << "\n";
14   std::cout << "max_size: " << myvector.max_size() << "\n";
15   return 0;
16 }
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

Member Functions Common to All Containers

- Examples
  - Default constructor
  - Several constructors with parameters
  - Destructor
- Class encapsulates data, operations on that data into a single unit
- Every container is a class
  - Several operations directly defined for a container
  - Provided as part of class definition

TABLE 4-5 Member functions common to all containers

Member function	Effect
Default constructor	Initializes the object to an empty state.
Constructor with parameters	In addition to the default constructor, every container has constructors with parameters. We describe these constructors when we discuss a specific container.
Copy constructor	Executes when an object is passed as a parameter by value, and when an object is declared and initialized using another object of the same type.
Destructor	Executes when the object goes out of scope.
`ct.empty()`	Returns `true` if container `ct` is empty and `false` otherwise.
`ct.size()`	Returns the number of elements currently in container `ct`.
`ct.max_size()`	Returns the maximum number of elements that can be inserted into container `ct`.
`ct1.swap(ct2)`	Swaps the elements of containers `ct1` and `ct2`.
`ct.begin()`	Returns an iterator to the first element into container `ct`.
`ct.end()`	Returns an iterator to the last element into container `ct`.
`ct.rbegin()`	Reverse begin. Returns a pointer to the last element into container `ct`. This function is used to process the elements of `ct` in reverse.
`ct.rend()`	Reverse end. Returns a pointer to the first element into container `ct`.
`ct.insert(position, elem)`	Inserts `elem` into container `ct` at the position specified by the argument `position`. Note that here `position` is an iterator.
`ct.erase(begin, end)`	Deletes all elements between `begin...end-1` from container `ct`.

One after the last element

One before the first element

TABLE 4-5 Member functions common to all containers (continued)

Member function	Effect
`ct.clear()`	Deletes all elements from the container. After a call to this function, container **ct** is empty.
Operator functions	
`ct1 = ct2`	Copies the elements of **ct2** into **ct1**. After this operation, the elements in both containers are the same.
`ct1 == ct2`	Returns **true** if containers **ct1** and **ct2** are equal and **false** otherwise.
`ct1 != ct2`	Returns **true** if containers **ct1** and **ct2** are not equal and **false** otherwise.
`ct1 < ct2`	Returns **true** if container **ct1** is less than container **ct2** and **false** otherwise.
`ct1 <= ct2`	Returns **true** if container **ct1** is less than or equal to container **ct2** and **false** otherwise.
`ct1 > ct2`	Returns **true** if container **ct1** is greater than container **ct2** and **false** otherwise.
`ct1 >= ct2`	Returns **true** if container **ct1** is greater than or equal to container **ct2** and **false** otherwise.

Member Functions Common to Sequence Containers

TABLE 4-6 Member functions common to all sequence containers

Expression	Effect
`seqCont.insert(position, elem)`	A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned.
`seqCont.insert(position, n, elem)`	`n` copies of `elem` are inserted at the position specified by `position`.
`seqCont.insert(position, beg, end)`	A copy of the elements, starting at `beg` until `end-1`, are inserted into `seqCont` at the position specified by `position`.
`seqCont.push_back(elem)`	A copy of `elem` is inserted into `seqCont` at the end.
`seqCont.pop_back()`	Deletes the last element.
`seqCont.erase(position)`	Deletes the element at the position specified by `position`.
`seqCont.erase(beg, end)`	Deletes all elements starting at `beg` until `end-1`.
`seqCont.clear()`	Deletes all elements from the container.
`seqCont.resize(num)`	Changes the number of elements to `num`. If `size()` grows, the new elements are created by their default constructor.
`seqCont.resize(num, elem)`	Changes the number of elements to `num`. If `size()` grows, the new elements are copies of `elem`.

The `copy` Algorithm

- Provides convenient way to output container elements
- Generic STL algorithm
  - Usable with any container type and arrays
- Does more than output container elements
  - Allows copying of elements from one place to another
- Function template copy definition
  - Contained in header file `algorithm`

`ostream` Iterator and Function `copy`

- Output container contents
  - Use a for loop and the function `begin`
    - Use the function `end` to set limit
  - Use Function `copy`
    - `ostream` iterator type specifies destination
- Creating an iterator of type `ostream`
  - Specify element type iterator will output
- Function `copy`
  - Can output container elements using `ostream` iterator
  - Directly specify `ostream` iterator in function `copy`

```
using namespace std;                                                    //Line 5

int main()                                                              //Line 6
{                                                                       //Line 7
    int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75};                  //Line 8

    vector<int> vecList(9);                                             //Line 9

    ostream_iterator<int> screen(cout, " ");                            //Line 10

    cout << "Line 11: intArray: ";                                      //Line 11
    copy(intArray, intArray + 9, screen);                               //Line 12
    cout << endl;                                                       //Line 13

    copy(intArray, intArray + 9, vecList.begin());                      //Line 14

    cout << "Line 15: vecList: ";                                       //Line 15
    copy(vecList.begin(), vecList.end(), screen);                       //Line 16
    cout << endl;                                                       //Line 17

    copy(intArray + 1, intArray + 9, intArray);                         //Line 18
    cout << "Line 19: After shifting the elements one "
         << "position to the left, intArray: " << endl;                 //Line 19
    copy(intArray, intArray + 9, screen);                               //Line 20
    cout << endl;                                                       //Line 21

    copy(vecList.rbegin() + 2, vecList.rend(),
                               vecList.rbegin());                       //Line 22
    cout << "Line 23: After shifting the elements down "
         << "by two positions, vecList:" << endl;                       //Line 23
    copy(vecList.begin(), vecList.end(), screen);                       //Line 24
    cout << endl;                                                       //Line 25

    return 0;                                                           //Line 26
}                                                                       //Line 27
```

Sequence Container: `deque`

- Deque: double-ended queue
- Implemented as dynamic arrays
  - Can expand in either direction
- Class name defining deque container
  - `deque`
- Header file `deque` contains
  - Definition of the class `deque`
  - Functions to implement various operations on a `deque` object
- Class `deque` contains several constructors

Sequence Container: `deque` (cont'd.)

TABLE 4-7 Various ways to declare a `deque` object

Statement	Effect
`deque<elementType> deq;`	Creates an empty `deque` container without any elements. (The default constructor is invoked.)
`deque<elementType> deq(otherDeq);`	Creates a deque container, `deq`, and initializes `deq` to the elements of `otherDeq`; `deq` and `otherDeq` are of the same type.
`deque<elementType> deq(size);`	Creates a deque container, `deq`, of size `size`. `deq` is initialized using the default constructor.
`deque<elementType> deq(n, elem);`	Creates a deque container, `deq`, of size `n`. `deq` is initialized using `n` copies of the element `elem`.
`deque<elementType> deq(begin, end);`	Creates a deque container, `deq`. `deq` is initialized to the elements in the range `[begin, end)`—that is, all elements in the range `begin...end-1`.

Sequence Container: `deque` (cont'd.)

TABLE 4-8 Various operations that can be performed on a `deque` object

Expression	Effect
`deq.assign(n,elem)`	Assigns `n` copies of `elem`.
`deq.assign(beg,end)`	Assigns all the elements in the range `beg...end-1`.
`deq.push_front(elem)`	Inserts `elem` at the beginning of `deq`.
`deq.pop_front()`	Removes the first element from `deq`.
`deq.at(index)`	Returns the element at the position specified by `index`.
`deq[index]`	Returns the element at the position specified by `index`.
`deq.front()`	Returns the first element. (Does not check whether the container is empty.)
`deq.back()`	Returns the last element. (Does not check whether the container is empty.)

STL Iterators

- Work like pointers
- Point to elements of a container (sequence or associative)
- Allow successive access to each container element
- Two most common operations on iterators
  - `++` (increment operator)
  - `*` (dereferencing operator)
- Examples

```
++cntItr;
*cntItr;
```

Types of Iterators

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random access iterators

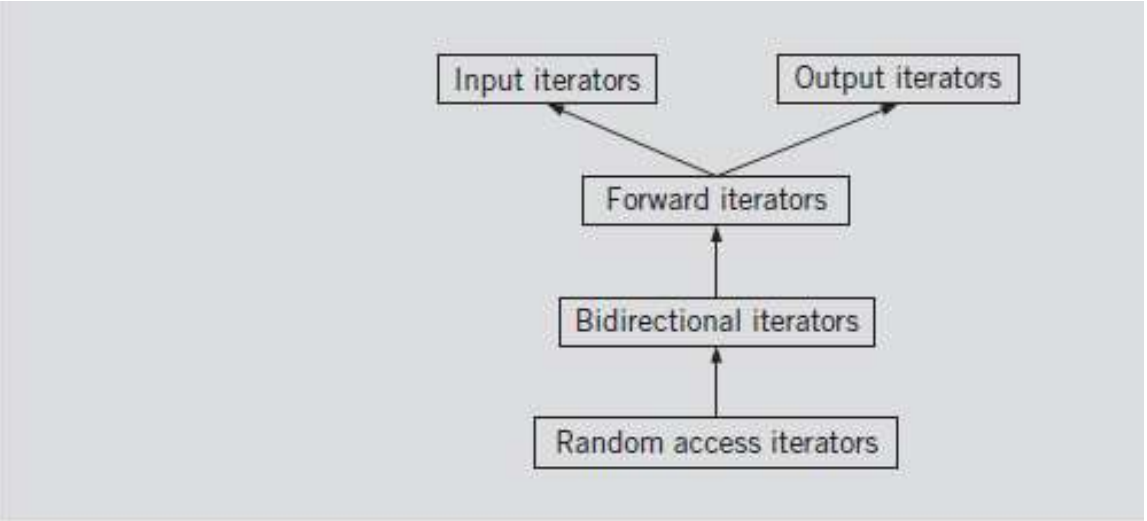

FIGURE 4-1 Iterator hierarchy

Input Iterators

- Read access
  - Step forward element-by-element
  - Return values element-by-element
- Provided for reading data from an input stream

Input Iterators (cont'd.)

TABLE 4-9 Operations on an input iterator

Expression	Effect
`*inputIterator`	Gives access to the element to which `inputIterator` points.
`inputIterator->member`	Gives access to the member of the element.
`++inputIterator`	Moves forward, returns the new position (preincrement).
`inputIterator++`	Moves forward, returns the old position (postincrement).
`inputIt1 == inputIt2`	Returns `true` if the two iterators are the same and `false` otherwise.
`inputIt1 != inputIt2`	Returns `true` if the two iterators are not the same and `false` otherwise.
`Type(inputIterator)`	Copies the iterators.

Output Iterators

- Write access
  - Step forward element-by-element
- Used for writing data to an output stream
- Cannot be used to iterate over a range twice
  - If we write data at the same position twice, there is no guarantee that the new value will replace the old value. *Hint: ostream_iterator*

Output Iterators (cont'd.)

TABLE 4-10 Operations on an output iterator

Expression	Effect
`*outputIterator = value;`	Writes the `value` at the position specified by the `outputIterator`.
`++outputIterator`	Moves forward, returns the new position (preincrement).
`outputIterator++`	Moves forward, returns the old position (postincrement).
`Type(outputIterator)`	Copies the iterators.

Forward Iterators

- Combination of
  - All of input iterators functionality and almost all output iterators functionality
- Can refer to same element in same collection
  - Can process same element more than once

Forward Iterators (cont'd.)

TABLE 4-11 Operations on a forward iterator

Expression	Effect
`*forwardIterator`	Gives access to the element to which `forwardIterator` points.
`forwardIterator->member`	Gives access to the member of the element.
`++forwardIterator`	Moves forward, returns the new position (preincrement).
`forwardIterator++`	Moves forward, returns the old position (postincrement).
`forwardIt1 == forwardIt2`	Returns `true` if the two iterators are the same and `false` otherwise.
`forwardIt1 != forwardIt2`	Returns `true` if the two iterators are not the same and `false` otherwise.
`forwardIt1 = forwardIt2`	Assignment.

Bidirectional Iterators

- Forward iterators that can also iterate backward over the elements
- Operations defined for forward iterators applicable to bidirectional Iterators
- To step backward
  - Decrement operations also defined for `biDirectionalIterator`
- Can be used only with containers of type:
  - `vector`, `deque`, `list`, `set`, `multiset`, `map`, and `multimap`

Bidirectional Iterators (cont'd.)

TABLE 4-12 Additional operations on a bidirectional iterator

Expression	Effect
`--biDirectionalIterator`	Moves backward, returns the new position (predecrement).
`biDirectionalIterator--`	Moves backward, returns the old position (postdecrement).

Random Access Iterators

- Bidirectional iterators that can randomly process container elements
- Can be used with containers of type:
  - `vector` , `deque` , `string`, and `arrays`
- Operations defined for bidirectional iterators applicable to random access iterators

Random Access Iterators (cont'd.)

TABLE 4-13 Additional operations on a random access iterator

Expression	Effect
`rAccessIterator[n]`	Accesses the nth element.
`rAccessIterator += n`	Moves `rAccessIterator` forward n elements if `n >= 0` and backward if `n < 0`.
`rAccessIterator -= n`	Moves `rAccessIterator` backward n elements if `n >= 0` and forward if `n < 0`.
`rAccessIterator + n`	Returns the iterator of the next nth element.
`n + rAccessIterator`	Returns the iterator of the next nth element.
`rAccessIterator - n`	Returns the iterator of the previous nth element.
`rAccessIt1 - rAccessIt2`	Returns the distance between the iterators `rAccessIt1` and `rAccessIt2`.
`rAccessIt1 < rAccessIt2`	Returns `true` if `rAccessIt1` is before `rAccessIt2` and `false` otherwise.
`rAccessIt1 <= rAccessIt2`	Returns `true` if `rAccessIt1` is before or equal to `rAccessIt2` and `false` otherwise.
`rAccessIt1 > rAccessIt2`	Returns `true` if `rAccessIt1` is after `rAccessIt2` and `false` otherwise.
`rAccessIt1 >= rAccessIt2`	Returns `true` if `rAccessIt1` is after or equal to `rAccessIt2` and `false` otherwise.

Iterators (cont'd.)

- `typedef iterator`
  - Every container (sequence or associative) contains a `typedef iterator`
  - Iterator into a container declared using `typedef iterator`
  - Must use appropriate container name, container element type, scope resolution operator

Iterators (cont'd.)

- `typedef const_iterator`
  - Modify container elements using an iterator into a container and dereferencing operator (*)
  - Prevents iterator from modifying elements of container declared as constant
  - Every container contains `typedef const_iterator`
  - Read-only iterator
- `typedef reverse_iterator`
  - Every container contains `typedef reverse_iterator`
  - Used to iterate through the elements of a container in reverse

Iterators (cont'd.)

- `typedef const_reverse iterator`
  - Read-only iterator
  - Used to iterate through elements of a container in reverse
  - Required if
    - Container declared as const
    - Need to iterate through the elements of the container in reverse

```cpp
// vector::begin/end
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```cpp
// vector::rbegin/rend
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (5);  // 5 default-constructed ints

  std::vector<int>::reverse_iterator rit = myvector.rbegin();

  int i=0;
  for (rit = myvector.rbegin(); rit!= myvector.rend(); ++rit)
    *rit = ++i;

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Iterators (cont'd.)

TABLE 4-14 Various `typedefs` common to all containers

typedef	Effect
`difference_type`	The type of result from subtracting two iterators referring to the same container.
`pointer`	A pointer to the type of elements stored in the container.
`reference`	A reference to the type of elements stored in the container.
`const_reference`	A constant reference to the type of elements stored in the container. A constant reference is read-only.
`size_type`	The type used to count the elements in a container. This type is also used to index through sequence containers, except `list` containers.
`value_type`	The type of container elements.

Stream Iterators

- `istream_iterator`
  - Used to input data into a program from an input stream
  - `class istream_iterator`
    - Contains definition of an input stream iterator
  - General syntax to use an `istream iterator`

```
istream_iterator<Type> isIdentifier(istream&);
```

Stream Iterators (cont'd.)

- `ostream iterators`
  - Used to output data from a program into an output stream
  - `class ostream_iterator`
    - Contains definition of an output stream iterator
  - General syntax to use an `ostream iterator`

```
ostream_iterator<Type> osIdentifier(ostream&);
```

or

```
ostream_iterator<Type> osIdentifier(ostream&, char* deLimit);
```

Summary

- STL
  - Provides class templates
    - Process lists, stacks, and queues
  - Three main components
    - Containers, iterators, and algorithms
  - STL containers: class templates
- Iterators
  - Step through the elements of a container
- Algorithms
  - Manipulate elements in a container

Summary (cont'd.)

- Main categories of containers
  - Sequence containers, associative containers, container adapters
- Three predefined sequence containers
  - `vector`, `deque`, and `list`
- `copy` algorithm
  - Copies elements in a given range to another place
- Function `copy`, using an `ostream` iterator
  - Can output the elements of a container
- Five categories of iterators: input, output, forward, bidirectional, random access iterator

Self Exercises

- Programming Exercises: 4, 8, 10