Chapter 15

Standard Library Containers and Iterators

OBJECTIVES

- ❑ Introduce the Standard Library containers, iterators.
- ❑ Use the vector, list and deque sequence containers.
- ❑ Use the stack, queue container adapters.

Topics

- ❑ **15.1 Introduction**
- ❑ **15.2 Introduction to Containers**
- ❑ **15.3 Introduction to Iterators**
- ❑ **15.4 Sequence Containers**
- ❑ **15.7 Container Adapters**

15.1 Introduction

- 标准模板库(*Standard Template Library*, STL)

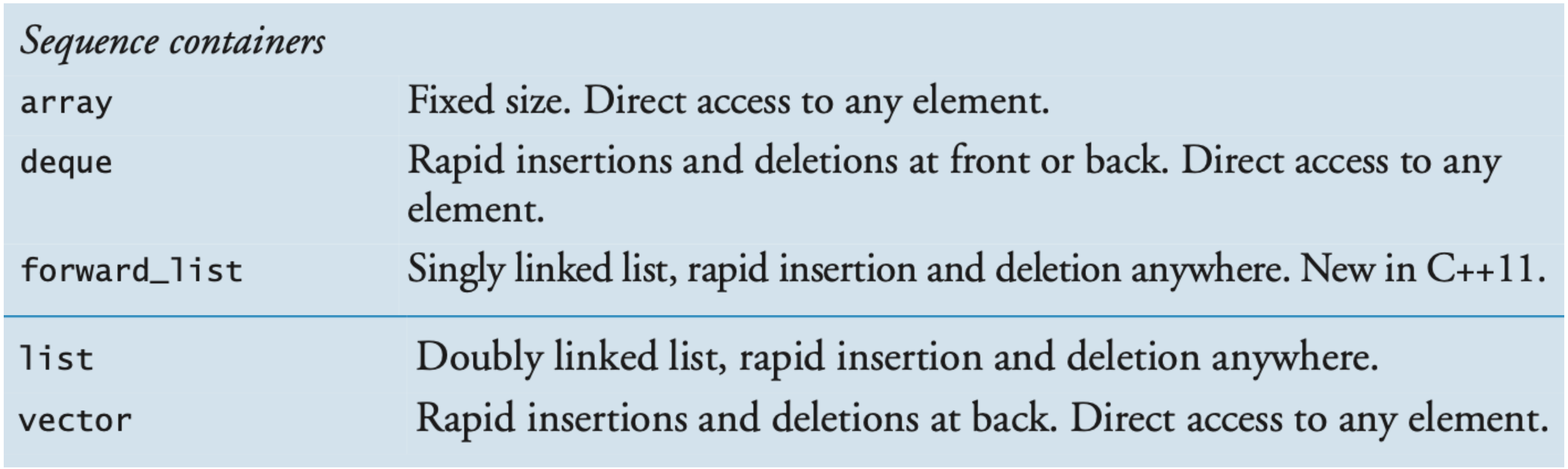
- Containers
- Iterators
- Algorithms

Topics

15.2 Introduction to Containers

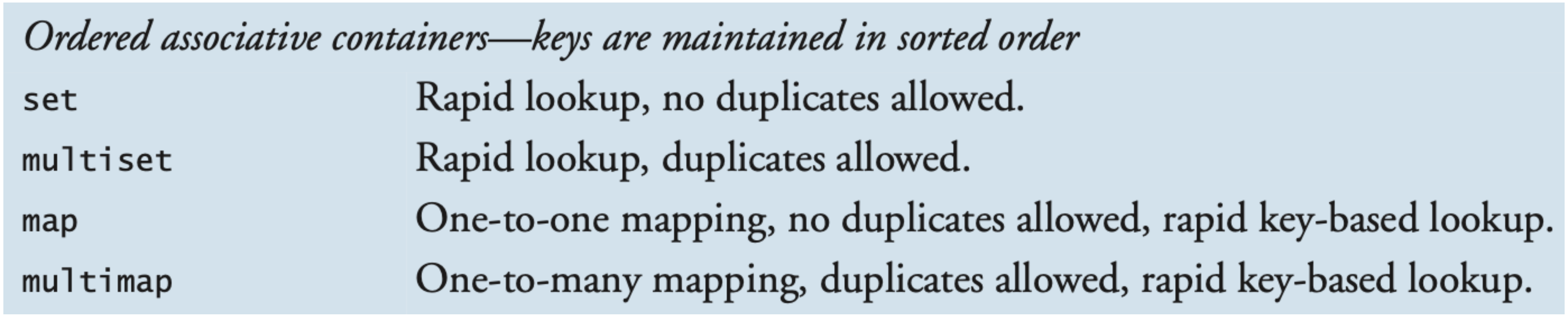
- Sequence containers
    - Array, deque, list, vector
- Ordered associative containers
    - Set, multiset, map, multimap
- Unordered associative containers
    - Unordered_set
- Container adapters
    - Stack, queue

15.2 Introduction to Containers

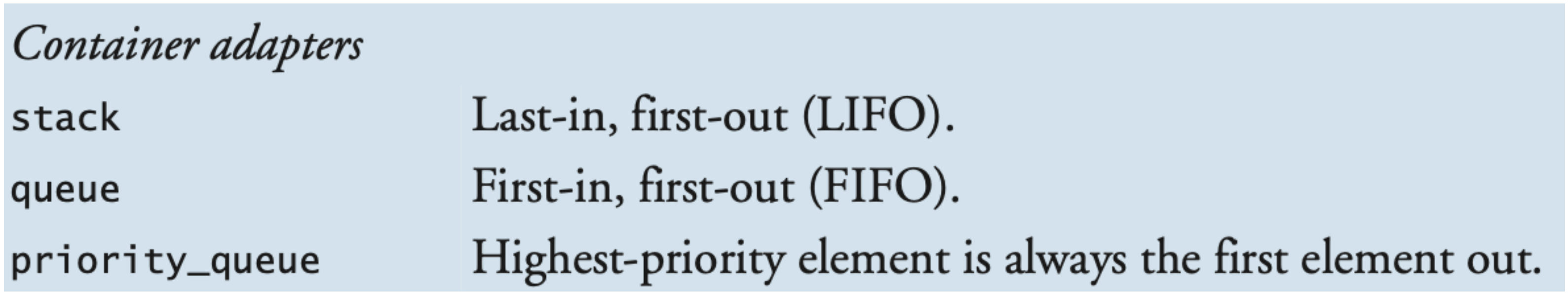


Sequence containers	
`array`	Fixed size. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`forward_list`	Singly linked list, rapid insertion and deletion anywhere. New in C++11.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
`vector`	Rapid insertions and deletions at back. Direct access to any element.

15.2 Introduction to Containers



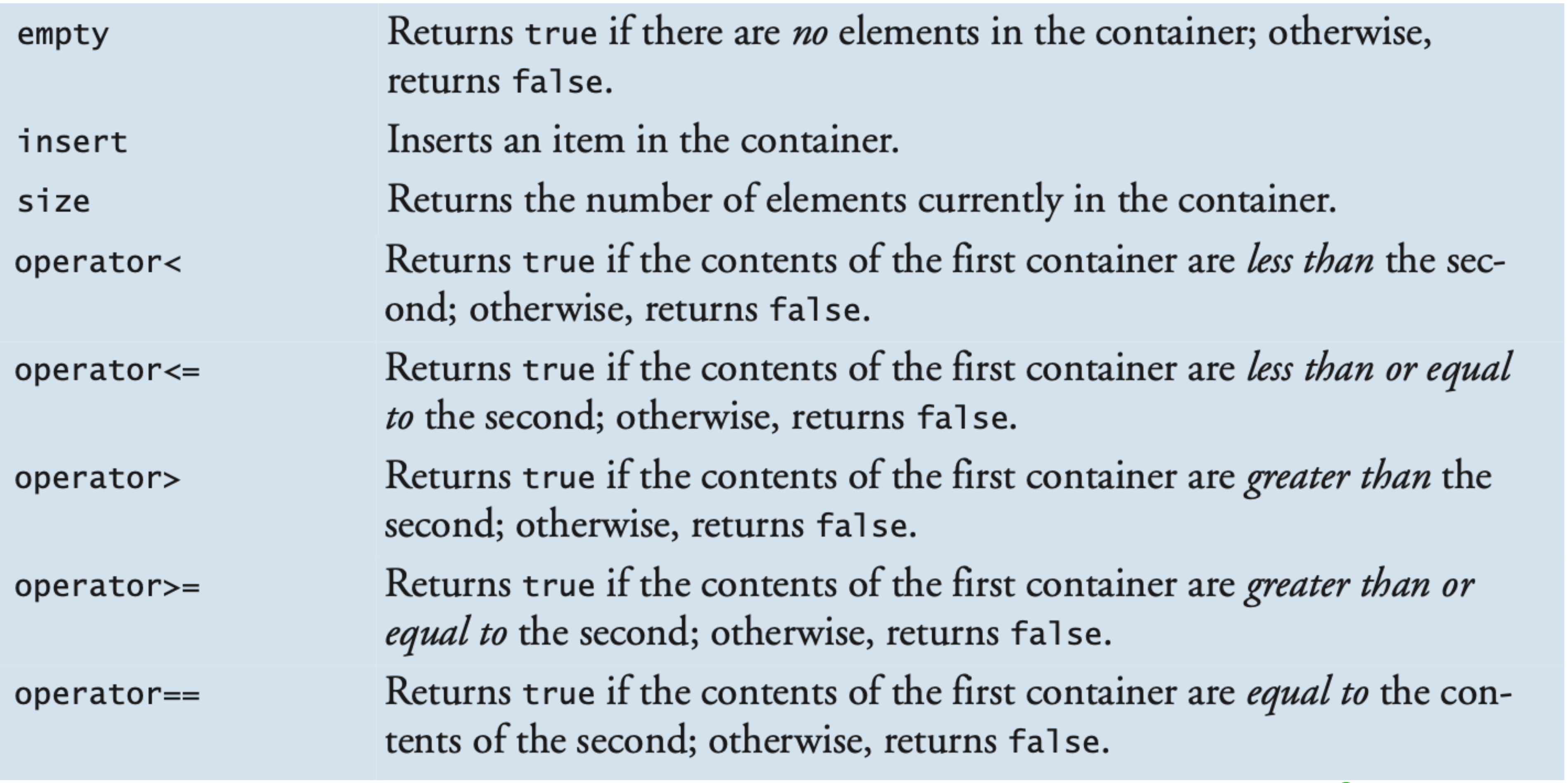
Ordered associative containers—keys are maintained in sorted order	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.



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.

15.2 Introduction to Containers

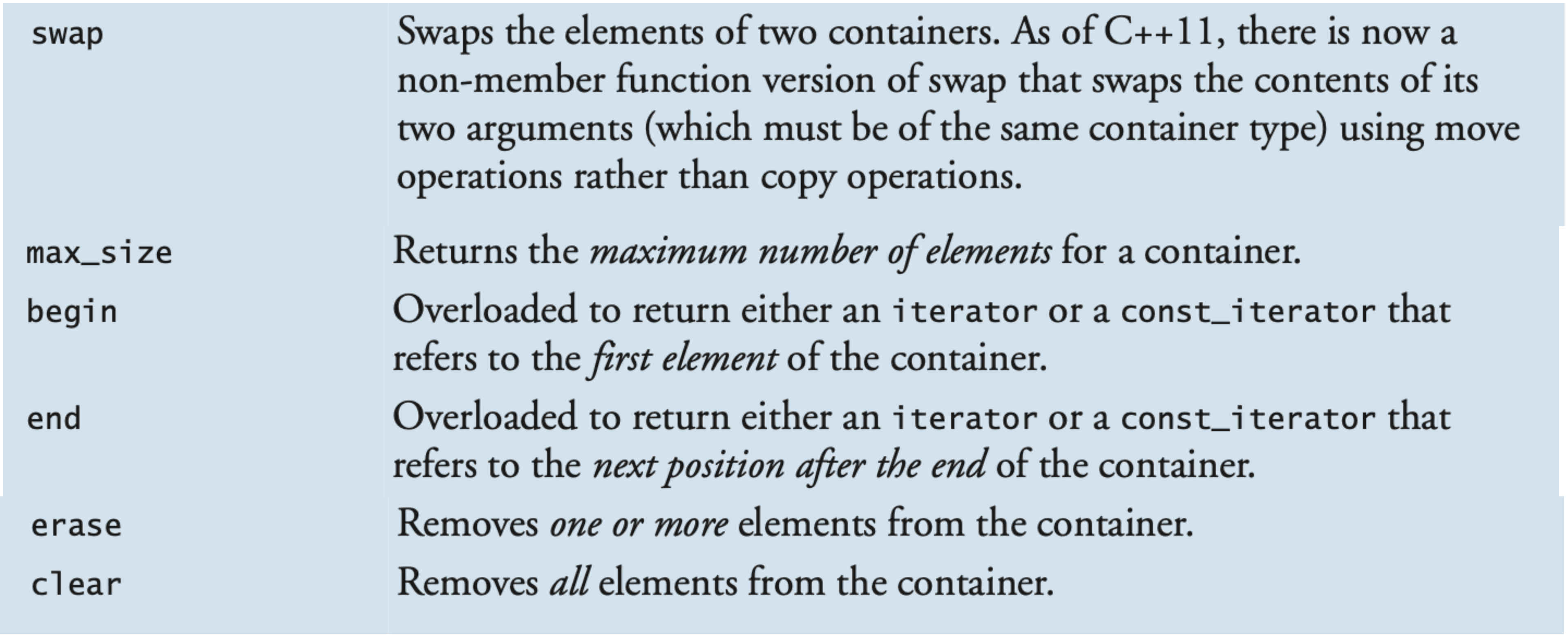
❑ Common Container Functions



`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<`	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`.

15.2 Introduction to Containers

❑Common Container Functions



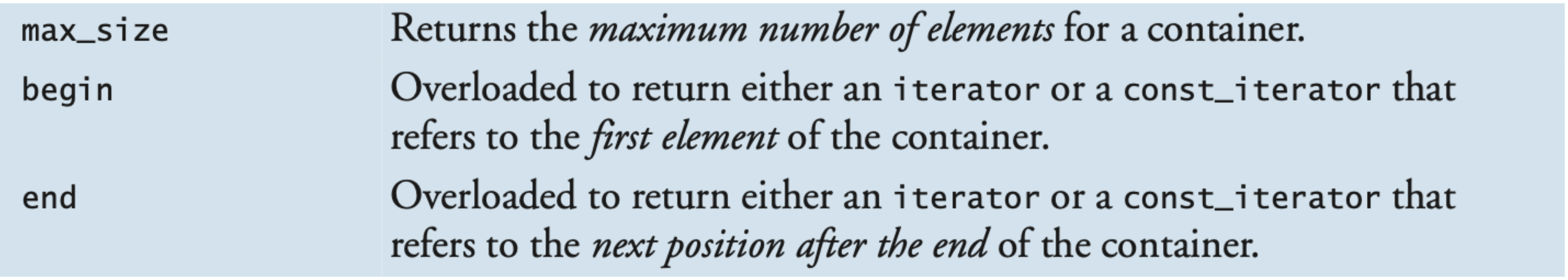
`swap`	Swaps the elements of two containers. As of C++11, there is now a non-member function version of swap that swaps the contents of its two arguments (which must be of the same container type) using move operations rather than copy operations.
`max_size`	Returns the *maximum number of elements* for 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.
`erase`	Removes *one or more* elements from the container.
`clear`	Removes *all* elements from the container.

Topics

- ❑ 15.1 Introduction
- ❑ 15.2 Introduction to Containers
- ❑ **15.3 Introduction to Iterators**
- ❑ 15.4 Sequence Containers
- ❑ 15.7 Container Adapters

15.3 Introduction to Iterators

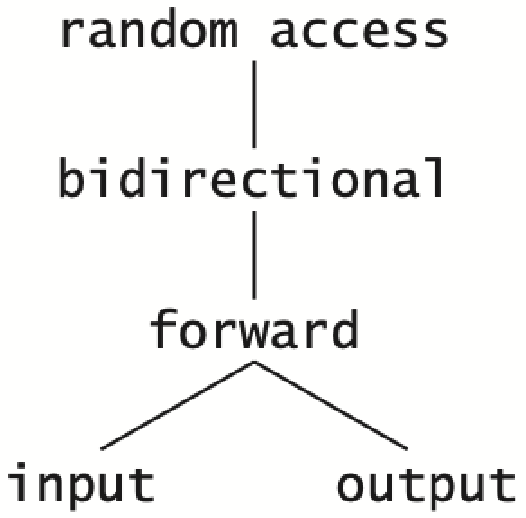
- **Iterators**
  - **Have many similarities to pointers**
  - **Dereference operator ***
  - **++ operator**

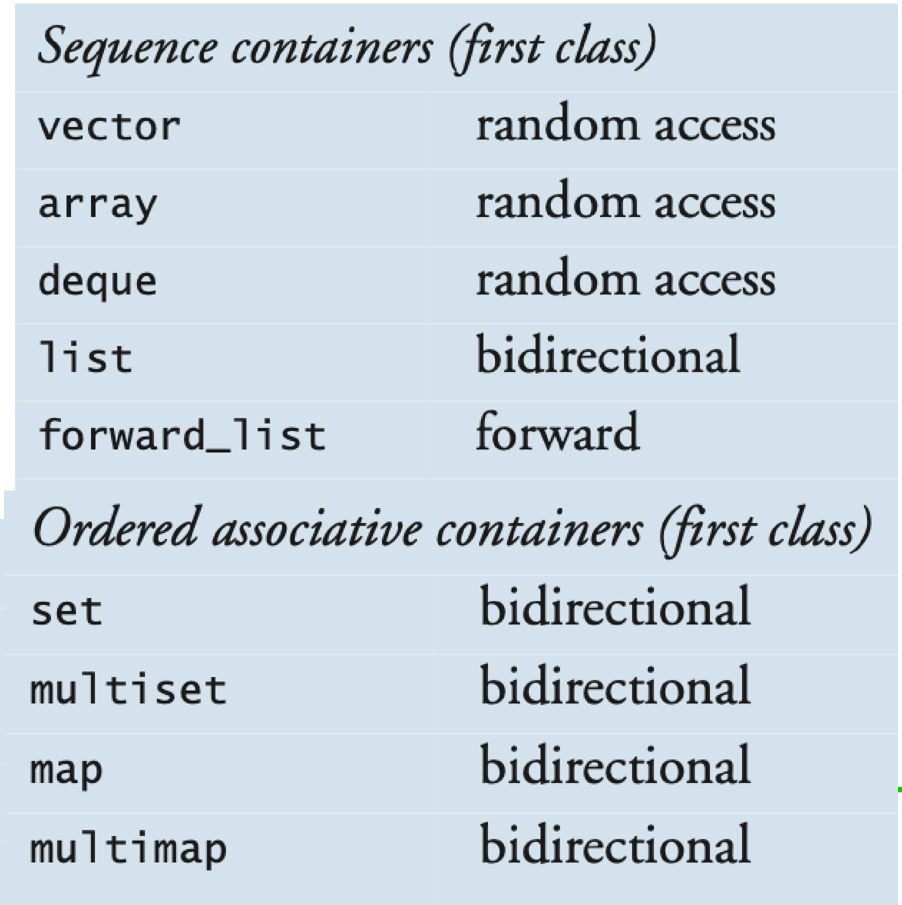


`max_size`	Returns the *maximum number of elements* for a container.
`begin`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *first element* of the container.
`end`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *next position after the end* of the container.

15.3 Introduction to Iterators

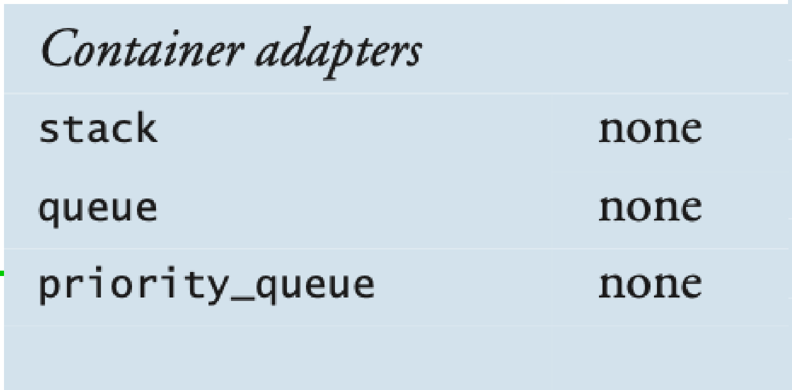
- Iterators category hierarchy





Sequence containers (first class)	
vector	random access
array	random access
deque	random access
list	bidirectional
forward_list	forward

Ordered associative containers (first class)	
set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional



Container adapters	
stack	none
queue	none
priority_queue	none

Topics

- ❑ 15.1 Introduction
- ❑ 15.2 Introduction to Containers
- ❑ 15.3 Introduction to Iterators
- ❑ **15.4 Sequence Containers**
- ❑ 15.7 Container Adapters

15.4 Sequence Containers -vector

- Class template *vector*
  - Contiguous memory locations
  - Direct access to any element
  - the number of elements need to grow

```cpp
int main()
{
   const size_t SIZE = 6; // define array size
   int values[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize values
   vector< int > integers; // create vector of ints

   cout << "The initial siz
      << "\nThe initial capa

   // function push_back is
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );

   cout << "\nThe size of integers is: " << integers.size()
```

```cpp
template < typename T > void printVector( const vector< T > &integers2 )
{
   // display vector elements using const_iterator
   for ( auto constIterator = integers2.cbegin();
      constIterator != integers2.cend(); ++constIterator )
      cout << *constIterator << ' ';
} // end function printVector
```

```cpp
   printVector( integers );
   cout << "\nReversed contents of vector integers: ";

   // display vector in reverse order using const_reverse_iterator
   for ( auto reverseIterator = integers.crbegin();
      reverseIterator!= integers.crend(); ++reverseIterator )
      cout << *reverseIterator << ' ';

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

```
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 built-in 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
```

```cpp
#include <array> // array class-template definition
#include <vector> // vector class-template definition
#include <algorithm> // copy algorithm
#include <iterato
#include <stdexce
using namespace s

int main()
{
   const size_t SIZE = 6;
   array< int, SIZE > values = { 1, 2, 3, 4, 5, 6 };
   vector< int > integers( values.cbegin(), values.cend() );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector integers contains: ";
   copy( integers.cbegin(), integers.cend(), output );

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

```
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
```

- 算法copy把integers中全部内容送到标准输出。复制容器中从第一个迭代器参数指定的元素一直到（不包括）第二个迭代器参数指定的元素之间的所有元素。

```
    try
    {
       integers.at(
    } // end try
    catch ( out_of_
    {
       cout << "\n\
    } // end catch

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

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

    // insert elements from the array values
    integers.insert( integers.cbegin(), values.cbegin(), values.cend() );
    cout << "\n\nContents of vector integers before clear: ";
    copy( integers.cbegin(), integers.cend(), output );

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

```
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
```

15.4 Sequence Containers -list

- Class template *list*
  - Allow insertion and deletion at any location rapidly
  - Support *bidirectional iterators*

15.

```
template < typename T > void printList( const list< T > &listRef )
{
   if ( listRef.empty() ) // list is empty
      cout << "List is empty";
   else
   {
      ostream_iterator< T > output( cout, " " );
      copy( listRef.cbegin(), listRef.cend(), output );
   } // end else
} // end function printList
```

```
int main(
{
   const
   array<
   list<
   list< int > otherValues; // create list of ints

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

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

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

   // insert elements of ints into otherValues
   otherValues.insert( otherValues.cbegin(), ints.cbegin(), ints.cend() );
   cout << "\nAfter insert, otherValues contains: ";
   printList( otherValues );
```

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
```

15.4 Sequ

```
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
```

```
// remove otherValues ele
values.splice( values.cend(), otherValues );
cout << "\nAfter splice, values contains: ";
printL

values
cout <
printList( values );

otherValues.insert( otherValues.cbegin(), ints.cbegin(), ints.cend() );
otherValues.sort(); // sort the list
cout << "\nAfter insert and sort, otherValues contains: ";
printList( otherValues );

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

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

- ❑ Splice函数删除otherValues中元素，并将其插入到第一个迭代器指定的位置之前。

- ❑ merge函数删除otherValues中元素，并将其按已排序的顺序插入到values中

15.4 Sequ

```
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
```

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

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

   // replace contents of values with elements of otherValues
   values.assign( otherValues.cbegin(), otherValues.cend() );
   cou
   pri

   //
   val
   cout << "\nAfter merge, values contains: ";
   printList( values );

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

- **assign函数用两个迭代器指定范围的元素取代原values里的内容。**

15.4 Sequence Containers -deque

- Class template *deque*
  - Allow indexed access, like a vector
  - Efficient insertion and deletion at its front and back, like a list
  - Support *random-access iterators*

iners

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

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

   cout << "values contains: ";

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

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

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

Topics

- 15.1 Introduction
- 15.2 Introduction to Containers
- 15.3 Introduction to Iterators
- 15.4 Sequence Containers
- **15.7 Container Adapters**

15.7 Container Adapters

- Stack
- Queue

```cpp
#include <stack> // stack adap
#include <vector> // vector cl
#include <list> // list class-
using namespace std;

// pushElements function-templ
template< typename T > void pu

// popElements function-templa
template< typename T > void po

int main()
{
    // stack with default under
    stack< int > intDequeStack;

    // stack with underlying ve
    stack< int, vector< int > >

    // stack with underlying li
    stack< int, list< int > > i

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

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

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

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

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

15.7 Container Adapters

- Stack
- Queue

```cpp
// Fig. 15.20: fig15_20.cpp
// Standard Library queue adapter class template.
#include <iostream>
#include <queue> // queue adapter definition
using namespace std;

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

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

   cout << "Popping from values: ";

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

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

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

Summary

- sequence containers
  - Vector
  - List
  - Deque
- container adapters
  - Stack
  - Queue