Illustrations List ***(Main Page)***

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. 20.1 Standard Library container classes.

Common member functions for all STL containers	Description
default constructor	A constructor to provide a default initialization of the container. Normally, each container has several constructors that provide a variety of 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`**.
`max_size`	Returns the maximum number of elements for a 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`** the first container is greater than the second container; otherwise, returns **`false`**.

Fig. 20.2 Common functions for all STL containers.

Common member functions for all STL containers	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 that are only found in first-class containers	
`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 position before the first element of the container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 20.2 Common functions for all STL containers.

.

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`**
`<bitset>`	

Fig. 20.3 Standard Library container header files.

typedef	Description
`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 only be used for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer to the type of element stored in the container.
`iterator`	An iterator that points to the type of element stored in the container.
`const_iterator`	A constant iterator that points to the type of element stored in the container and can only be used 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 to the type of element stored in the container and can only be used 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. 20.4 Common `typedef`s found in first-class containers.

```
// Fig. 20.5: fig20_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>
#include <iterator>

using namespace std;

int main()
{
   cout << "Enter two integers: ";

```

Fig. 20.5 Demonstrating input and output stream iterators (part 1 of 2).

```
   istream_iterator< int > inputInt( cin );
   int number1, number2;

   number1 = *inputInt;  // read first int from standard input
   ++inputInt;           // move iterator to next input value
   number2 = *inputInt;  // read next int from standard input

   cout << "The sum is: ";

   ostream_iterator< int > outputInt( cout );

   *outputInt = number1 + number2;  // output result to cout
```

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

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

Fig. 20.5 Demonstrating input and output stream iterators (part 2 of 2).

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 of the container) 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 input iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retain 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 of the container). Forward iterators support multi-pass 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. 20.6 Iterator categories.

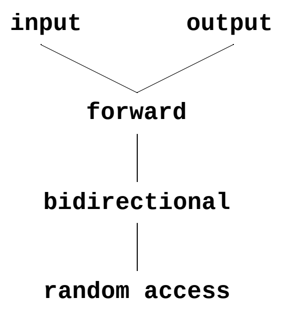

Fig. 20.7 Iterator category hierarchy.

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

Fig. 20.8 Iterator types supported by each Standard Library container.

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. 20.9 Predefined iterator **typedef**s.

Iterator operation	Description
All iterators	
++p	preincrement an iterator
p++	postincrement an iterator
Input iterators	
***p**	dereference an iterator for use as an *rvalue*
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 (for use as an *lvalue*)

Fig. 20.10 Iterator operations for each type of iterator.

Iterator operation	Description
p = p1	assign one iterator to another
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
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	Results in an iterator positioned at **p** incremented by **i** positions.
p - i	Results in an iterator positioned at **p** decremented by **i** positions.
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. 20.10 Iterator operations for each type of iterator.

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

Fig. 20.11 Mutating-sequence algorithms.

Non-mutating sequence algorithms		
`adjacent-find()`	`equal()`	`mismatch()`
`count()`	`find()`	`search()`
`count_if()`	`for_each()`	`search_n()`

Fig. 20.12 Non-mutating sequence algorithms.

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

Fig. 20.13 Numerical algorithms from header file `<numeric>`.

```
// Fig. 20.14: fig20_14.cpp
// Testing Standard Library vector class template
#include <iostream>
#include <vector>

using namespace std;

template < class T >
void printVector( const vector< T > &vec );

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v;

   cout << "The initial size of v is: " << v.size()
        << "\nThe initial capacity of v is: " << v.capacity();
   v.push_back( 2 );  // method push_back() is in
   v.push_back( 3 );  // every sequence container
   v.push_back( 4 );
   cout << "\nThe size of v is: " << v.size()
        << "\nThe capacity of v is: " << v.capacity();
   cout << "\n\nContents of array a using pointer notation: ";

   for ( int *ptr = a; ptr != a + SIZE; ++ptr )
      cout << *ptr << ' ';

   cout << "\nContents of vector v using iterator notation: ";
   printVector( v );

   cout << "\nReversed contents of vector v: ";

   vector< int >::reverse_iterator p2;

   for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
      cout << *p2 << ' ';
   cout << endl;
   return 0;
}
```

Fig. 20.14 Demonstrating Standard Library `vector` class template (part 1 of 2).

```

template < class T >
void printVector( const vector< T > &vec )
{
   vector< T >::const_iterator p1;

   for ( p1 = vec.begin(); p1 != vec.end(); ++p1 )
      cout << *p1 << ' ';
}
```

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

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2
```

Fig. 20.14 Demonstrating Standard Library `vector` class template (part 2 of 2).

```
// Fig. 20.15: fig20_15.cpp
// Testing Standard Library vector class template
// element-manipulation functions
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v( a, a + SIZE );
   ostream_iterator< int > output( cout, " " );
   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output );

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

   v[ 0 ] = 7;          // set first element to 7
   v.at( 2 ) = 10;      // set element at position 2 to 10
   v.insert( v.begin() + 1, 22 );  // insert 22 as 2nd element
```

Fig. 20.15 Demonstrating Standard Library `vector` class template element-manipulation functions (part 1 of 2).

```
   cout << "\nContents of vector v after changes: ";
   copy( v.begin(), v.end(), output );

   try {
      v.at( 100 ) = 777;   // access element out of range
   }
   catch ( out_of_range e ) {
      cout << "\nException: " << e.what();
   }

   v.erase( v.begin() );
   cout << "\nContents of vector v after erase: ";
   copy( v.begin(), v.end(), output );
   v.erase( v.begin(), v.end() );
   cout << "\nAfter erase, vector v "
        << ( v.empty() ? "is" : "is not" ) << " empty";

   v.insert( v.begin(), a, a + SIZE );
   cout << "\nContents of vector v before clear: ";
   copy( v.begin(), v.end(), output );
   v.clear();  // clear calls erase to empty a collection
   cout << "\nAfter clear, vector v "
        << ( v.empty() ? "is" : "is not" ) << " empty";

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

```
Vector v contains: 1 2 3 4 5 6
First element of v: 1
Last element of v: 6
Contents of vector v after changes: 7 22 2 10 4 5 6
Exception: invalid vector<T> subscript
Contents of vector v after erase: 22 2 10 4 5 6
After erase, vector v is empty
Contents of vector v before clear: 1 2 3 4 5 6
After clear, vector v is empty
```

Fig. 20.15 Demonstrating Standard Library **`vector`** class template element-manipulation functions (part 2 of 2).

STL exception types	Description
`out_of_range`	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to **`vector`** member function **`at`**.
`invalid_argument`	Indicates an invalid argument was passed to a function.
`length_error`	Indicates an attempt to create too long a container, **`string`**, etc.
`bad_alloc`	Indicates that an attempt to allocate memory with **`new`** (or with an allocator) failed because not enough memory was available.

Fig. 20.16 STL exception types.

```
// Fig. 20.17: fig20_17.cpp
// Testing Standard Library class list
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template < class T >
void printList( const list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int a[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values, otherValues;

   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

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

   otherValues.insert( otherValues.begin(), a, a + SIZE );
```

Fig. 20.17 Demonstrating Standard Library `list` class template (part 1 of 3).

```
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.splice( values.end(), otherValues );
   cout << "\nAfter splice values contains: ";
   printList( values );

   values.sort();
   cout << "\nvalues contains: ";
   printList( values );
   otherValues.insert( otherValues.begin(), a, a + SIZE );
   otherValues.sort();
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.merge( otherValues );
   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front();
   values.pop_back();   // all sequence containers
   cout << "\nAfter pop_front and pop_back values contains:\n";
   printList( values );

   values.unique();
   cout << "\nAfter unique values contains: ";
   printList( values );

   // method swap is available in all containers
   values.swap( otherValues );
   cout << "\nAfter swap:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.assign( otherValues.begin(), otherValues.end() );
   cout << "\nAfter assign values contains: ";
   printList( values );

   values.merge( otherValues );
   cout << "\nvalues contains: ";
   printList( values );
   values.remove( 4 );
   cout << "\nAfter remove( 4 ) values contains: ";
   printList( values );
   cout << endl;
   return 0;
}

```

Fig. 20.17 Demonstrating Standard Library **list** class template (part 2 of 3).

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

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
otherValues contains: 2 6 4 8
After splice values contains: 1 2 3 4 2 6 4 8
values contains: 1 2 2 3 4 4 6 8
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 8
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
values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ) values contains: 2 2 3 3 6 6 8 8
```

Fig. 20.17 Demonstrating Standard Library **list** class template (part 3 of 3).

```
// Fig. 20.18: fig20_18.cpp
// Testing Standard Library class deque
#include <iostream>
#include <deque>
#include <algorithm>

using namespace std;

int main()
{
   deque< double > values;
   ostream_iterator< double > output( cout, " " );

   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );

   cout << "values contains: ";

   for ( int i = 0; i < values.size(); ++i )
      cout << values[ i ] << ' ';

   values.pop_front();
   cout << "\nAfter pop_front values contains: ";
   copy ( values.begin(), values.end(), output );

   values[ 1 ] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4 values contains: ";
   copy ( values.begin(), values.end(), output );
   cout << endl;
   return 0;
}
```

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

Fig. 20.18 Demonstrating Standard Library **deque** class template.

```
// Fig. 20.19: fig20_19.cpp
// Testing Standard Library class multiset
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   typedef multiset< int, less< int > > ims;
   ims intMultiset;    // ims for "integer multiset"
   ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
        << " values of 15 in the multiset\n";
   intMultiset.insert( 15 );
   intMultiset.insert( 15 );
   cout << "After inserts, there are "
```

```
          << intMultiset.count( 15 )
          << " values of 15 in the multiset\n";

     ims::const_iterator result;

     result = intMultiset.find( 15 );  // find returns iterator

     if ( result != intMultiset.end() ) // if iterator not at end
        cout << "Found value 15\n";     // found search value 15

     result = intMultiset.find( 20 );

     if ( result == intMultiset.end() )    // will be true hence
        cout << "Did not find value 20\n"; // did not find 20

     intMultiset.insert( a, a + SIZE ); // add array a to multiset
     cout << "After insert intMultiset contains:\n";
     copy( intMultiset.begin(), intMultiset.end(), output );

     cout << "\nLower bound of 22: "
          << *( intMultiset.lower_bound( 22 ) );
     cout << "\nUpper bound of 22: "
          << *( intMultiset.upper_bound( 22 ) );

     pair< ims::const_iterator, ims::const_iterator > p;

     p = intMultiset.equal_range( 22 );
     cout << "\nUsing equal_range of 22"
          << "\n   Lower bound: " << *( p.first )
          << "\n   Upper bound: " << *( p.second );
```

Fig. 20.19 Demonstrating Standard Library **multiset** class template (part 1 of 2).

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

```
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
Using equal_range of 22
   Lower bound: 22
   Upper bound: 30
```

Fig. 20.19 Demonstrating Standard Library **multiset** class template (part 2 of 2).

```
// Fig. 20.20: fig20_20.cpp
// Testing Standard Library class set
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
   typedef set< double, less< double > > double_set;
   const int SIZE = 5;
   double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
   double_set doubleSet( a, a + SIZE );;
   ostream_iterator< double > output( cout, " " );

   cout << "doubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   pair< double_set::const_iterator, bool > p;

   p = doubleSet.insert( 13.8 ); // value not in set
   cout << '\n' << *( p.first )
        << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   p = doubleSet.insert( 9.5 );  // value already in set
   cout << '\n' << *( p.first )
        << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

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

```
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. 20.20 Demonstrating Standard Library `set` class template.

```
// Fig. 20.21: fig20_21.cpp
// Testing Standard Library class multimap
#include <iostream>
#include <map>

using namespace std;

int main()
{
   typedef multimap< int, double, less< int > > mmid;
   mmid pairs;

   cout << "There are currently " << pairs.count( 15 )
        << " pairs with key 15 in the multimap\n";
   pairs.insert( mmid::value_type( 15, 2.7 ) );
   pairs.insert( mmid::value_type( 15, 99.3 ) );
   cout << "After inserts, there are "
        << pairs.count( 15 )
        << " pairs with key 15\n";
   pairs.insert( mmid::value_type( 30, 111.11 ) );
   pairs.insert( mmid::value_type( 10, 22.22 ) );
   pairs.insert( mmid::value_type( 25, 33.333 ) );
   pairs.insert( mmid::value_type( 20, 9.345 ) );
   pairs.insert( mmid::value_type( 5, 77.54 ) );
   cout << "Multimap pairs contains:\nKey\tValue\n";

   for ( mmid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

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

Fig. 20.21 Demonstrating Standard Library `multimap` class template (part 1 of 2).

```
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. 20.21 Demonstrating Standard Library `multimap` class template (part 2 of 2).

```
// Fig. 20.22: fig20_22.cpp
// Testing Standard Library class map
#include <iostream>
#include <map>

using namespace std;

int main()
{
   typedef map< int, double, less< int > > mid;
   mid pairs;

   pairs.insert( mid::value_type( 15, 2.7 ) );
   pairs.insert( mid::value_type( 30, 111.11 ) );
   pairs.insert( mid::value_type( 5, 1010.1 ) );
   pairs.insert( mid::value_type( 10, 22.22 ) );
   pairs.insert( mid::value_type( 25, 33.333 ) );
   pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored
   pairs.insert( mid::value_type( 20, 9.345 ) );
   pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored
   cout << "pairs contains:\nKey\tValue\n";

   mid::const_iterator iter;

   for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   pairs[ 25 ] = 9999.99;  // change existing value for 25
   pairs[ 40 ] = 8765.43;  // insert new value for 40
   cout << "\nAfter subscript operations, pairs contains:"
        << "\nKey\tValue\n";

   for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

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

Fig. 20.22 Demonstrating Standard Library **map** class template (part 1 of 2).

```
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. 20.22 Demonstrating Standard Library **map** class template (part 2 of 2).

```
// Fig. 20.23: fig20_23.cpp
// Testing Standard Library class stack
#include <iostream>
#include <stack>
#include <vector>
#include <list>

using namespace std;

template< class T >
void popElements( T &s );

```

```
int main()
{
   stack< int > intDequeStack; // default is deque-based stack
   stack< int, vector< int > > intVectorStack;
   stack< int, list< int > > intListStack;

   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );
   }

   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );

   cout << endl;
   return 0;
}

```

Fig. 20.23 Demonstrating Standard Library `stack` adapter class (part 1 of 2).

```
template< class T >
void popElements( T &s )
{
   while ( !s.empty() ) {
      cout << s.top() << ' ';
      s.pop();
   }
}
```

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

Fig. 20.23 Demonstrating Standard Library `stack` adapter class (part 2 of 2).

```
// Fig. 20.24: fig20_24.cpp
// Testing Standard Library adapter class template queue
#include <iostream>
#include <queue>

using namespace std;

int main()
{
   queue< double > values;

   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";

   while ( !values.empty() ) {
      cout << values.front() << ' ';   // does not remove
      values.pop();                    // removes element
   }

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

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

Fig. 20.24 Demonstrating Standard Library **queue** adapter class templates.

```
// Fig. 20.25: fig20_25.cpp
// Testing Standard Library class priority_queue
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

int main()
{
   priority_queue< double > priorities;

   priorities.push( 3.2 );
   priorities.push( 9.8 );
   priorities.push( 5.4 );

   cout << "Popping from priorities: ";

   while ( !priorities.empty() ) {
      cout << priorities.top() << ' ';
      priorities.pop();
   }

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

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

Fig. 20.25 Demonstrating Standard Library **priority_queue** adapter class.

```
// Fig. 20.26: fig20_26.cpp
// Demonstrating fill, fill_n, generate, and generate_n
// Standard Library methods.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

char nextLetter();

int main()
{
   vector< char > chars( 10 );
   ostream_iterator< char > output( cout, " " );

   fill( chars.begin(), chars.end(), '5' );
   cout << "Vector chars after filling with 5s:\n";
   copy( chars.begin(), chars.end(), output );

   fill_n( chars.begin(), 5, 'A' );
   cout << "\nVector chars after filling five elements"
        << " with As:\n";
   copy( chars.begin(), chars.end(), output );

   generate( chars.begin(), chars.end(), nextLetter );
   cout << "\nVector chars after generating letters A-J:\n";
   copy( chars.begin(), chars.end(), output );
```

Fig. 20.26 Demonstrating Standard Library functions **fill**, **fill_n**, **generate** and **generate_n** (part 1 of 2).

```

   generate_n( chars.begin(), 5, nextLetter );
   cout << "\nVector chars after generating K-O for the"
        << " first five elements:\n";
   copy( chars.begin(), chars.end(), output );

   cout << endl;
   return 0;
}

char nextLetter()
{
   static char letter = 'A';
   return letter++;
}
```

```
Vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:
A B C D E F G H I J

Vector chars after generating K-O for the first five el-
ements:
K L M N O F G H I J
```

Fig. 20.26 Demonstrating Standard Library functions `fill`, `fill_n`, `generate` and `generate_n` (part 2 of 2).

```
// Fig. 20.27: fig20_27.cpp
// Demonstrates Standard Library functions equal,
// mismatch, lexicographical_compare.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
   vector< int > v1( a1, a1 + SIZE ),
                 v2( a1, a1 + SIZE ),
                 v3( a2, a2 + SIZE );
   ostream_iterator< int > output( cout, " " );

```

Fig. 20.27 Demonstrating Standard Library functions `equal`, `mismatch` and `lexicographical_compare` (part 1 of 2).

```
   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );
   cout << "\nVector v3 contains: ";
   copy( v3.begin(), v3.end(), output );

   bool result = equal( v1.begin(), v1.end(), v2.begin() );
   cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v2.\n";

   result = equal( v1.begin(), v1.end(), v3.begin() );
   cout << "Vector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v3.\n";

   pair< vector< int >::iterator,
         vector< int >::iterator > location;
   location = mismatch( v1.begin(), v1.end(), v3.begin() );
   cout << "\nThere is a mismatch between v1 and v3 at "
```

```
        << "location " << ( location.first - v1.begin() )
        << "\nwhere v1 contains " << *location.first
        << " and v3 contains " << *location.second
        << "\n\n";

   char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";

   result =
      lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
   cout << c1
        << ( result ? " is less than " : " is greater than " )
        << c2;

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

```
Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v2 contains 1000

HELLO is greater than BYE BYE
```

Fig. 20.27 Demonstrating Standard Library functions `equal`, `mismatch` and `lexicographical_compare` (part 2 of 2).

```
// Fig. 20.28: fig20_28.cpp
// Demonstrates Standard Library functions remove, remove_if
// remove_copy and remove_copy_if
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater9( int );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   // Remove 10 from v
   vector< int > v( a, a + SIZE );
   vector< int >::iterator newLastElement;
   cout << "Vector v before removing all 10s:\n";
   copy( v.begin(), v.end(), output );
   newLastElement = remove( v.begin(), v.end(), 10 );
   cout << "\nVector v after removing all 10s:\n";
   copy( v.begin(), newLastElement, output );

   // Copy from v2 to c, removing 10s
   vector< int > v2( a, a + SIZE );
   vector< int > c( SIZE, 0 );
   cout << "\n\nVector v2 before removing all 10s "
        << "and copying:\n";
   copy( v2.begin(), v2.end(), output );
   remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
   cout << "\nVector c after removing all 10s from v2:\n";
   copy( c.begin(), c.end(), output );

```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 1 of 3).

```
   // Remove elements greater than 9 from v3
   vector< int > v3( a, a + SIZE );
   cout << "\n\nVector v3 before removing all elements"
        << "\ngreater than 9:\n";
   copy( v3.begin(), v3.end(), output );
   newLastElement = remove_if( v3.begin(), v3.end(),
                               greater9 );
   cout << "\nVector v3 after removing all elements"
        << "\ngreater than 9:\n";
   copy( v3.begin(), newLastElement, output );

   // Copy elements from v4 to c,
   // removing elements greater than 9
   vector< int > v4( a, a + SIZE );
   vector< int > c2( SIZE, 0 );
   cout << "\n\nVector v4 before removing all elements"
        << "\ngreater than 9 and copying:\n";
   copy( v4.begin(), v4.end(), output );
   remove_copy_if( v4.begin(), v4.end(),
                   c2.begin(), greater9 );
   cout << "\nVector c2 after removing all elements"
        << "\ngreater than 9 from v4:\n";
   copy( c2.begin(), c2.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int x )
{
   return x > 9;
}
```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 2 of 3).

```
Vector v before removing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v after removing all 10s:
2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c after removing all 10s from v2:
2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements
greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after removing all elements
greater than 9:
2 4 6 8

Vector v4 before removing all elements
greater than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after removing all elements
greater than 9 from v4:
2 4 6 8 0 0 0 0 0 0
```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 3 of 3).

```
// Fig. 20.29: fig20_29.cpp
// Demonstrates Standard Library functions replace, replace_if
// replace_copy and replace_copy_if
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater9( int );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   // Replace 10s in v1 with 100
   vector< int > v1( a, a + SIZE );
   cout << "Vector v1 before replacing all 10s:\n";
   copy( v1.begin(), v1.end(), output );
   replace( v1.begin(), v1.end(), 10, 100 );
   cout << "\nVector v1 after replacing all 10s with 100s:\n";
   copy( v1.begin(), v1.end(), output );

```

Fig. 20.29 Demonstrating Standard Library functions **replace**, **replace_if**, **replace_copy** and **replace_copy_if** (part 1 of 3).

```
   // copy from v2 to c1, replacing 10s with 100s
   vector< int > v2( a, a + SIZE );
   vector< int > c1( SIZE );
   cout << "\n\nVector v2 before replacing all 10s "
        << "and copying:\n";
   copy( v2.begin(), v2.end(), output );
   replace_copy( v2.begin(), v2.end(),
                 c1.begin(), 10, 100 );
   cout << "\nVector c1 after replacing all 10s in v2:\n";
   copy( c1.begin(), c1.end(), output );

   // Replace values greater than 9 in v3 with 100
   vector< int > v3( a, a + SIZE );
   cout << "\n\nVector v3 before replacing values greater"
        << " than 9:\n";
   copy( v3.begin(), v3.end(), output );
   replace_if( v3.begin(), v3.end(), greater9, 100 );
   cout << "\nVector v3 after replacing all values greater"
        << "\nthan 9 with 100s:\n";
   copy( v3.begin(), v3.end(), output );

   // Copy v4 to c2, replacing elements greater than 9 with 100
   vector< int > v4( a, a + SIZE );
   vector< int > c2( SIZE );
   cout << "\n\nVector v4 before replacing all values greater"
        << "\nthan 9 and copying:\n";
   copy( v4.begin(), v4.end(), output );
   replace_copy_if( v4.begin(), v4.end(), c2.begin(),
                    greater9, 100 );
   cout << "\nVector c2 after replacing all values greater"
        << "\nthan 9 in v4:\n";
   copy( c2.begin(), c2.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int x )
{
   return x > 9;
}
```

Fig. 20.29 Demonstrating Standard Library functions **`replace`**, **`replace_if`**, **`replace_copy`** and **`replace_copy_if`** (part 2 of 3).

```
Vector v1 before replacing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing all 10s with 100s:
100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater
than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater
than 9 in v4:
100 2 100 4 100 6 100 8 100 100
```

Fig. 20.29 Demonstrating Standard Library functions **replace**, **replace_if**, **replace_copy** and **replace_copy_if** (part 3 of 3).

```
// Fig. 20.30: fig20_30.cpp
// Examples of mathematical algorithms in the Standard Library.
#include <iostream>
#include <algorithm>
#include <numeric>     // accumulate is defined here
#include <vector>

using namespace std;

bool greater9( int );
void outputSquare( int );
int calculateCube( int );

int main()
{
   const int SIZE = 10;
   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before random_shuffle: ";
   copy( v.begin(), v.end(), output );
   random_shuffle( v.begin(), v.end() );
   cout << "\nVector v after random_shuffle: ";
   copy( v.begin(), v.end(), output );

   int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   vector< int > v2( a2, a2 + SIZE );
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 1 of 3).

```
   cout << "\n\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );
   int result = count( v2.begin(), v2.end(), 8 );
   cout << "\nNumber of elements matching 8: " << result;

   result = count_if( v2.begin(), v2.end(), greater9 );
   cout << "\nNumber of elements greater than 9: " << result;

   cout << "\n\nMinimum element in Vector v2 is: "
        << *( min_element( v2.begin(), v2.end() ) );

   cout << "\nMaximum element in Vector v2 is: "
        << *( max_element( v2.begin(), v2.end() ) );

   cout << "\n\nThe total of the elements in Vector v is: "
        << accumulate( v.begin(), v.end(), 0 );

   cout << "\n\nThe square of every integer in Vector v is:\n";
   for_each( v.begin(), v.end(), outputSquare );

   vector< int > cubes( SIZE );
   transform( v.begin(), v.end(), cubes.begin(),
              calculateCube );
   cout << "\n\nThe cube of every integer in Vector v is:\n";
   copy( cubes.begin(), cubes.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int value ) { return value > 9; }

void outputSquare( int value ) { cout << value * value << ' '; }

int calculateCube( int value ) { return value * value * value; }
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 2 of 3).

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 3 of 3).

```
// Fig. 20.31: fig20_31.cpp
// Demonstrates search and sort capabilities.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater10( int value );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
   vector< int > v( a, a + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output );

   vector< int >::iterator location;
   location = find( v.begin(), v.end(), 16 );

   if ( location != v.end() )
      cout << "\n\nFound 16 at location "
           << ( location - v.begin() );
   else
      cout << "\n\n16 not found";

   location = find( v.begin(), v.end(), 100 );

   if ( location != v.end() )
      cout << "\nFound 100 at location "
           << ( location - v.begin() );
   else
      cout << "\n100 not found";

   location = find_if( v.begin(), v.end(), greater10 );

   if ( location != v.end() )
      cout << "\n\nThe first value greater than 10 is "
           << *location << "\nfound at location "
           << ( location - v.begin() );
   else
      cout << "\n\nNo values greater than 10 were found";

   sort( v.begin(), v.end() );
   cout << "\n\nVector v after sort: ";
   copy( v.begin(), v.end(), output );

```

Fig. 20.31 Basic searching and sorting algorithms of the Standard Library (part 1 of 2).

```
   if ( binary_search( v.begin(), v.end(), 13 ) )
      cout << "\n\n13 was found in v";
   else
      cout << "\n\n13 was not found in v";

   if ( binary_search( v.begin(), v.end(), 100 ) )
      cout << "\n100 was found in v";
   else
      cout << "\n100 was not found in v";

   cout << endl;
   return 0;
}

bool greater10( int value ) { return value > 10; }
```

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

Fig. 20.31 Basic searching and sorting algorithms of the Standard Library (part 2 of 2).

```
// Fig. 20.32: fig20_32.cpp
// Demonstrates iter_swap, swap and swap_ranges.
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   ostream_iterator< int > output( cout, " " );

   cout << "Array a contains:\n";
   copy( a, a + SIZE, output );
```

Fig. 20.32 Demonstrating `swap`, `iter_swap` and `swap_ranges` (part 1 of 2).

```

   swap( a[ 0 ], a[ 1 ] );
   cout << "\nArray a after swapping a[0] and a[1] "
        << "using swap:\n";
   copy( a, a + SIZE, output );

   iter_swap( &a[ 0 ], &a[ 1 ] );
   cout << "\nArray a after swapping a[0] and a[1] "
        << "using iter_swap:\n";
   copy( a, a + SIZE, output );

   swap_ranges( a, a + 5, a + 5 );
   cout << "\nArray a after swapping the first five elements\n"
        << "with the last five elements:\n";
   copy( a, a + SIZE, output );

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

```
Array a contains:
1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
6 7 8 9 10 1 2 3 4 5
```

Fig. 20.32 Demonstrating `swap`, `iter_swap` and `swap_ranges` (part 2 of 2).

```
// Fig. 20.33: fig20_33.cpp
// Demonstrates miscellaneous functions: copy_backward, merge,
// unique and reverse.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
   int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE );
   vector< int > v2( a2, a2 + SIZE );

   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );

   vector< int > results( v1.size() );
   copy_backward( v1.begin(), v1.end(), results.end() );
   cout << "\n\nAfter copy_backward, results contains: ";
   copy( results.begin(), results.end(), output );

   vector< int > results2( v1.size() * v2.size() );
   merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
          results2.begin() );
   cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
   copy( results2.begin(), results2.end(), output );

   vector< int >::iterator endLocation;
   endLocation = unique( results2.begin(), results2.end() );
```

Fig. 20.33 Demonstrating `copy_backward`, `merge`, `unique` and `reverse` (part 2 of 2).

```
   cout << "\n\nAfter unique results2 contains:\n";
   copy( results2.begin(), endLocation, output );

   cout << "\n\nVector v1 after reverse: ";
   reverse( v1.begin(), v1.end() );
   copy( v1.begin(), v1.end(), output );

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

```
Vector v1 contains: 1 3 5 7 9
Vector v2 contains: 2 4 5 7 9

After copy_backward results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1
```

Fig. 20.33 Demonstrating `copy_backward`, `merge`, `unique` and `reverse` (part 2 of 2).

```
// Fig. 20.34: fig20_34.cpp
// Demonstrates miscellaneous functions: inplace_merge,
// reverse_copy, and unique_copy.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE );

   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );

   inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
   cout << "\nAfter inplace_merge, v1 contains: ";
   copy( v1.begin(), v1.end(), output );

   vector< int > results1;
   unique_copy( v1.begin(), v1.end(),
                back_inserter( results1 ) );
   cout << "\nAfter unique_copy results1 contains: ";
   copy( results1.begin(), results1.end(), output );

   vector< int > results2;
   cout << "\nAfter reverse_copy, results2 contains: ";
   reverse_copy( v1.begin(), v1.end(),
                 back_inserter( results2 ) );
   copy( results2.begin(), results2.end(), output );

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

```
Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

Fig. 20.34 Demonstrating `inplace_merge`, `unique_copy` and `reverse_copy`.

```
// Fig. 20.35: fig20_35.cpp
// Demonstrates includes, set_difference, set_intersection,
// set_symmetric_difference and set_union.
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
   int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
   int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
   ostream_iterator< int > output( cout, " " );

   cout << "a1 contains: ";
   copy( a1, a1 + SIZE1, output );
   cout << "\na2 contains: ";
   copy( a2, a2 + SIZE2, output );
   cout << "\na3 contains: ";
   copy( a3, a3 + SIZE2, output );

   if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
      cout << "\na1 includes a2";
   else
      cout << "\na1 does not include a2";

   if ( includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
      cout << "\na1 includes a3";
   else
      cout << "\na1 does not include a3";

   int difference[ SIZE1 ];
   int *ptr = set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2,
                              difference );
   cout << "\nset_difference of a1 and a2 is: ";
   copy( difference, ptr, output );

   int intersection[ SIZE1 ];
   ptr = set_intersection( a1, a1 + SIZE1, a2, a2 + SIZE2,
                           intersection );
```

Fig. 20.35 Demonstrating `set` operations of the Standard Library (part 1 of 2).

```
   cout << "\nset_intersection of a1 and a2 is: ";
   copy( intersection, ptr, output );

   int symmetric_difference[ SIZE1 ];
   ptr = set_symmetric_difference( a1, a1 + SIZE1,
                 a2, a2 + SIZE2, symmetric_difference );
   cout << "\nset_symmetric_difference of a1 and a2 is: ";
   copy( symmetric_difference, ptr, output );

   int unionSet[ SIZE3 ];
   ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
   cout << "\nset_union of a1 and a3 is: ";
   copy( unionSet, ptr, output );
   cout << endl;
   return 0;
}
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
a1 includes a2
a1 does not include a3
set_difference of a1 and a2 is: 1 2 3 9 10
set_intersection of a1 and a2 is: 4 5 6 7 8
set_symmetric_difference of a1 and a2 is: 1 2 3 9 10
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

Fig. 20.35 Demonstrating `set` operations of the Standard Library (part 2 of 2).

```
// Fig. 20.36: fig20_36.cpp
// Demonstrates lower_bound, upper_bound and equal_range for
// a sorted sequence of values.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains:\n";
   copy( v.begin(), v.end(), output );

   vector< int >::iterator lower;
   lower = lower_bound( v.begin(), v.end(), 6 );
   cout << "\n\nLower bound of 6 is element "
        << ( lower - v.begin() ) << " of vector v";

   vector< int >::iterator upper;
   upper = upper_bound( v.begin(), v.end(), 6 );
   cout << "\nUpper bound of 6 is element "
        << ( upper - v.begin() ) << " of vector v";

   pair< vector< int >::iterator, vector< int >::iterator > eq;
   eq = equal_range( v.begin(), v.end(), 6 );
   cout << "\nUsing equal_range:\n"
        << "   Lower bound of 6 is element "
        << ( eq.first - v.begin() ) << " of vector v";
   cout << "\n   Upper bound of 6 is element "
        << ( eq.second - v.begin() ) << " of vector v";

```

```
    cout << "\n\nUse lower_bound to locate the first point\n"
         << "at which 5 can be inserted in order";
    lower = lower_bound( v.begin(), v.end(), 5 );
    cout << "\n   Lower bound of 5 is element "
         << ( lower - v.begin() ) << " of vector v";

    cout << "\n\nUse upper_bound to locate the last point\n"
         << "at which 7 can be inserted in order";
    upper = upper_bound( v.begin(), v.end(), 7 );
    cout << "\n   Upper bound of 7 is element "
         << ( upper - v.begin() ) << " of vector v";

```

Fig. 20.36 Demonstrating **lower_bound**, **upper_bound** and **equal_range** (part 1 of 2).

```
    cout << "\n\nUse equal_range to locate the first and\n"
         << "last point at which 5 can be inserted in order";
    eq = equal_range( v.begin(), v.end(), 5 );
    cout << "\n   Lower bound of 5 is element "
         << ( eq.first - v.begin() ) << " of vector v";
    cout << "\n   Upper bound of 5 is element "
         << ( eq.second - v.begin() ) << " of vector v"
         << endl;
    return 0;
}
```

```
Vector v contains:
2 2 4 4 4 6 6 6 6 8

Lower bound of 6 is element 5 of vector v
Upper bound of 6 is element 9 of vector v
Using equal_range:
   Lower bound of 6 is element 5 of vector v
   Upper bound of 6 is element 9 of vector v

Use lower_bound to locate the first point
at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v

Use upper_bound to locate the last point
at which 7 can be inserted in order
   Upper bound of 7 is element 9 of vector v

Use equal_range to locate the first and
last point at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v
   Upper bound of 5 is element 5 of vector v
```

Fig. 20.36 Demonstrating **lower_bound**, **upper_bound** and **equal_range** (part 2 of 2).

```
// Fig. 20.37: fig20_37.cpp
// Demonstrating push_heap, pop_heap, make_heap and sort_heap.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   int i;
```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 1 of 3).

```
   vector< int > v( a, a + SIZE ), v2;
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before make_heap:\n";
   copy( v.begin(), v.end(), output );
   make_heap( v.begin(), v.end() );
   cout << "\nVector v after make_heap:\n";
   copy( v.begin(), v.end(), output );
   sort_heap( v.begin(), v.end() );
   cout << "\nVector v after sort_heap:\n";
   copy( v.begin(), v.end(), output );

   // perform the heapsort with push_heap and pop_heap
   cout << "\n\nArray a contains: ";
   copy( a, a + SIZE, output );

   for ( i = 0; i < SIZE; ++i ) {
      v2.push_back( a[ i ] );
      push_heap( v2.begin(), v2.end() );
      cout << "\nv2 after push_heap(a[" << i << "]): ";
      copy( v2.begin(), v2.end(), output );
   }

   for ( i = 0; i < v2.size(); ++i ) {
      cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
      pop_heap( v2.begin(), v2.end() - i );
      copy( v2.begin(), v2.end(), output );
   }

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

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 2 of 3).

```
Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40
v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22
v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100
```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 3 of 3).

```cpp
// Fig. 20.38: fig20_38.cpp
// Demonstrating min and max
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
   cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
   cout << "\nThe minimum of 'G' and 'Z' is: "
        << min( 'G', 'Z' );
   cout << "\nThe maximum of 'G' and 'Z' is: "
        << max( 'G', 'Z' ) << endl;
   return 0;
}
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

Fig. 20.38 Demonstrating algorithms **`min`** and **`max`**.

Algorithm	Description
`adjacent_difference`	Beginning with the second element in a sequence, calculate the difference (using operator **`-`**) between the current element and the previous element, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third output iterator argument indicates where the results should be stored. A second version of this function takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
`inner_product`	This function calculates the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
`partial_sum`	Calculate a running total (using operator **`+`**) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third argument (an output iterator) indicates where the results should be stored. A second version of this function takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.
`nth_element`	This function uses three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this function executes, all elements to the left of the partitioning element are less than that element and all elements to the right of the partitioning element are greater than or equal to that element. A second version of this function takes as a fourth argument a binary comparison function.
`partition`	This function is similar to **`nth_element`**; however, it requires less powerful bidirectional iterators, so it is more flexible than **`nth_element`**. Function **`partition`** requires two bidirectional iterators indicating the range of elements to partition. The third element is a unary predicate function that helps partition the elements such that all elements in the sequence for which the predicate is **`true`** are to the left (toward the beginning of the sequence) of all elements for which the predicate is **`false`**. A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns **`false`**.

Fig. 20.39 Algorithms not covered in this chapter.

Algorithm	Description
`stable_partition`	This function is similar to **`partition`** except that elements for which the predicate function returns **`true`** are maintained in their original order and elements for which the **`predicate`** function returns **`false`** are maintained in their original order.
`next_permutation`	Next lexicographical permutation of a sequence.
`prev_permutation`	Previous lexicographical permutation of a sequence.
`rotate`	This function takes three forward iterator arguments and rotates the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.
`rotate_copy`	This function is identical to **`rotate`** except that the results are stored in a separate sequence indicated by the fourth argument—an output iterator. The two sequences must be the same number of elements.
`adjacent_find`	This function returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the **`end`** of the sequence.
`partial_sort`	This function uses three random-access iterators to sort part of a sequence. The first and last arguments indicate the entire sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator **`<`** (a binary predicate function can also be supplied). The elements from the second argument iterator to the end of the sequence are in an undefined order.
`partial_sort_copy`	This function uses two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator **`<`** (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
`stable_sort`	The function is similar to **`sort`** except that all equal elements are maintained in their original order.

Fig. 20.39 Algorithms not covered in this chapter.

```cpp
// Fig. 20.40: fig20_40.cpp
// Using a bitset to demonstrate the Sieve of Eratosthenes.
#include <iostream>
#include <iomanip>
#include <bitset>
#include <cmath>

using namespace std;

int main()
{
   const int size = 1024;
   int i, value, counter;
   bitset< size > sieve;

   sieve.flip();

```

Fig. 20.40 Demonstrating class **`bitset`** and the Sieve of Eratosthenes (part 1 of 3).

```cpp
   // perform Sieve of Eratosthenes
   int finalBit = sqrt( sieve.size() ) + 1;

   for ( i = 2; i < finalBit; ++i )
      if ( sieve.test( i ) )
         for ( int j = 2 * i; j < size; j += i )
            sieve.reset( j );

   cout << "The prime numbers in the range 2 to 1023 are:\n";

   for ( i = 2, counter = 0; i < size; ++i )
      if ( sieve.test( i ) ) {
         cout << setw( 5 ) << i;

         if ( ++counter % 12 == 0 )
            cout << '\n';
      }

   cout << endl;

   // get a value from the user to determine if it is prime
   cout << "\nEnter a value from 1 to 1023 (-1 to end): ";
   cin >> value;

   while ( value != -1 ) {
      if ( sieve[ value ] )
         cout << value << " is a prime number\n";
      else
         cout << value << " is not a prime number\n";

      cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
      cin >> value;
   }

   return 0;
}
```

Fig. 20.40 Demonstrating class **`bitset`** and the Sieve of Eratosthenes (part 2 of 3).

```
The prime numbers in the range 2 to 1023 are:
    2    3    5    7   11   13   17   19   23   29   31   37
   41   43   47   53   59   61   67   71   73   79   83   89
   97  101  103  107  109  113  127  131  137  139  149  151
  157  163  167  173  179  181  191  193  197  199  211  223
  227  229  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349  353  359
  367  373  379  383  389  397  401  409  419  421  431  433
  439  443  449  457  461  463  467  479  487  491  499  503
  509  521  523  541  547  557  563  569  571  577  587  593
  599  601  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733  739  743
  751  757  761  769  773  787  797  809  811  821  823  827
  829  839  853  857  859  863  877  881  883  887  907  911
  919  929  937  941  947  953  967  971  977  983  991  997
 1009 1013 1019 1021

Enter a value from 1 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1
```

Fig. 20.40 Demonstrating class **`bitset`** and the Sieve of Eratosthenes (part 3 of 3).

STL function objects	Type
`divides< T >`	arithmetic
`equal_to< T >`	relational
`greater< T >`	relational
`greater_equal< T >`	relational
`less< T >`	relational
`less_equal< T >`	relational
`logical_and< T >`	logical
`logical_not< T >`	logical
`logical_or< T >`	logical
`minus< T >`	arithmetic
`modulus< T >`	arithmetic
`negate< T >`	arithmetic
`not_equal_to< T >`	relational
`plus< T >`	arithmetic
`multiplies< T >`	arithmetic

Fig. 20.41 Function objects in the Standard Library .

```cpp
// Fig. 20.42: fig20_42.cpp
// Demonstrating function objects.
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>

using namespace std;

// binary function adds the square of its second argument and
// the running total in its first argument and
// returns the sum
int sumSquares( int total, int value )
   { return total + value * value; }
```

Fig. 20.42 Demonstrating a binary function object (part 1 of 2).

```cpp

// binary function class template which defines an overloaded
// operator() that function adds the square of its second
// argument and the running total in its first argument and
// returns the sum
template< class T >
class SumSquaresClass : public binary_function< T, T, T >
{
public:
   const T &operator()( const T &total, const T &value )
      { return total + value * value; }
};

int main()
{
   const int SIZE = 10;
   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );
   int result = 0;

   cout << "vector v contains:\n";
   copy( v.begin(), v.end(), output );
   result = accumulate( v.begin(), v.end(), 0, sumSquares );
   cout << "\n\nSum of squares of elements in vector v using "
        << "binary\nfunction sumSquares: " << result;

   result = accumulate( v.begin(), v.end(), 0,
                        SumSquaresClass< int >() );
   cout << "\n\nSum of squares of elements in vector v using "
        << "binary\nfunction object of type "
        << "SumSquaresClass< int >: " << result << endl;
   return 0;
}
```

```
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in vector v using binary
function sumSquares: 385

Sum of squares of elements in vector v using binary
function object of type SumSquaresClass< int >: 385
```

Fig. 20.42 Demonstrating a binary function object (part 2 of 2).