IS 0020
Program Design and Software Tools

Standard Template Library

Lecture 12

April 6, 2004

Introduction to the Standard Template Library (STL)

- STL
  - Powerful, template-based components
    - Containers: template data structures
    - Iterators: like pointers, access elements of containers
    - Algorithms: data manipulation, searching, sorting, etc.
  - Object- oriented programming: reuse, reuse, reuse
  - Only an introduction to STL, a huge class library

21.1.1 Introduction to Containers

- Three types of containers
  - Sequence containers
    - Linear data structures (vectors, linked lists)
    - First-class container
  - Associative containers
    - Non-linear, can find elements quickly
    - Key/value pairs
    - First-class container
  - Container adapters
    - Near containers
    - Similar to containers, with reduced functionality
- Containers have some common functions

STL Container Classes (Fig. 21.1)

- Sequence containers
  - `vector`
  - `deque`
  - `list`
- Associative containers
  - `set`
  - `multiset`
  - `map`
  - `multimap`
- Container adapters
  - `stack`
  - `queue`
  - `priority_queue`

Common STL Member Functions (Fig. 21.2)

- Member functions for all containers
  - Default constructor, copy constructor, destructor
  - `empty`
  - `max_size`, `size`
  - `= < <= > >= == !=`
  - `swap`
- Functions for first-class containers
  - `begin`, `end`
  - `rbegin`, `rend`
  - `erase`, `clear`

Common STL typedefs (Fig. 21.4)

- **`typedef`**s for first-class containers
  - **`value_type`**
  - **`reference`**
  - **`const_reference`**
  - **`pointer`**
  - **`iterator`**
  - **`const_iterator`**
  - **`reverse_iterator`**
  - **`const_reverse_iterator`**
  - **`difference_type`**
  - **`size_type`**

21.1.2 Introduction to Iterators

- Iterators similar to pointers
  - Point to first element in a container
  - Iterator operators same for all containers
    - **`*`** dereferences
    - **`++`** points to next element
    - **`begin()`** returns iterator to first element
    - **`end()`** returns iterator to last element
  - Use iterators with sequences (ranges)
    - Containers
    - Input sequences: **`istream_iterator`**
    - Output sequences: **`ostream_iterator`**

21.1.2 Introduction to Iterators

- Usage
  - **`std::istream_iterator< int > inputInt( cin )`**
    - Can read input from **`cin`**
    - **`*inputInt`**
      - Dereference to read first **`int`** from **`cin`**
    - **`++inputInt`**
      - Go to next **`int`** in stream
  - **`std::ostream_iterator< int > outputInt(cout)`**
    - Can output **`int`**s to **`cout`**
    - **`*outputInt = 7`**
      - Outputs **`7`** to **`cout`**
    - **`++outputInt`**
      - Advances iterator so we can output next **`int`**

fig21_05.cpp
(1 of 2)

```
// Fig. 21.5: fig21_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iterator>  // ostream_

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

   // create istream_iterator for reading int values from cin
   std::istream_iterator< int > inputInt( cin );

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

```

Note creation of **`istream_iterator`**. For compilation reasons, we use **`std::`** rather than a **`using`** statement.

Access and assign the iterator like a pointer.

Outline

fig21_05.cpp
(2 of 2)

fig21_05.cpp
output (1 of 1)

```cpp
   // create ostream_iterator for writing int values to cout
   std::ostream_iterator< int > outputInt( cout );

   cout << "The sum is: ";
   *outputInt = number1 + number2;  // output result to cout
   cout << endl;

   return 0;

} // end main
```

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

Create an **ostream_iterator** is similar. Assigning to this iterator outputs to **cout**.

Iterator Categories (Fig. 21.6)

- Input
  - Read elements from container, can only move forward
- Output
  - Write elements to container, only forward
- Forward
  - Combines input and output, retains position
  - Multi-pass (can pass through sequence twice)
- Bidirectional
  - Like forward, but can move backwards as well
- Random access
  - Like bidirectional, but can also jump to any element

Iterator Types Supported (Fig. 21.8)

- Sequence containers
  - **`vector`**: random access
  - **`deque`**: random access
  - **`list`**: bidirectional
- Associative containers (all bidirectional)
  - **`set`**
  - **`multiset`**
  - **`Map`**
  - **`multimap`**
- Container adapters (no iterators supported)
  - **`stack`**
  - **`queue`**
  - **`priority_queue`**

Iterator Operations (Fig. 21.10)

- All
  - `++p`, `p++`
- Input iterators
  - `*p`
  - `p = p1`
  - `p == p1`, `p != p1`
- Output iterators
  - `*p`
  - `p = p1`
- Forward iterators
  - Have functionality of input and output iterators

Iterator Operations (Fig. 21.10)

- Bidirectional
  - `--p`, `p--`
- Random access
  - `p + i`, `p += i`
  - `p - i`, `p -= i`
  - `p[i]`
  - `p < p1`, `p <= p1`
  - `p > p1`, `p >= p1`

21.1.3 Introduction to Algorithms

- STL has algorithms used generically across containers
  - Operate on elements indirectly via iterators
  - Often operate on sequences of elements
    - Defined by pairs of iterators
    - First and last element
  - Algorithms often return iterators
    - **`find()`**
    - Returns iterator to element, or **`end()`** if not found
  - Premade algorithms save programmers time and effort

21.2 Sequence Containers

- Three sequence containers
  - **`vector`** - based on arrays
  - **`deque`** - based on arrays
  - **`list`** - robust linked list

21.2.1 vector Sequence Container

- **`vector`**
  - **`<vector>`**
  - Data structure with contiguous memory locations
    - Access elements with **`[]`**
  - Use when data must be sorted and easily accessible
- When memory exhausted
  - Allocates larger, contiguous area of memory
  - Copies itself there
  - Deallocates old memory
- Has random access iterators

21.2.1 vector Sequence Container

- Declarations
  - **`std::vector <type> v;`**
    - ***type***: **`int`**, **`float`**, etc.
- Iterators
  - **`std::vector<type>::const_iterator iterVar;`**
    - **`const_iterator`** cannot modify elements
  - **`std::vector<type>::reverse_iterator iterVar;`**
    - Visits elements in reverse order (end to beginning)
    - Use **`rbegin`** to get starting point
    - Use **`rend`** to get ending point

21.2.1 vector Sequence Container

- **`vector`** functions
  - **`v.push_back(value)`**
    - Add element to end (found in all sequence containers).
  - **`v.size()`**
    - Current size of vector
  - **`v.capacity()`**
    - How much vector can hold before reallocating memory
    - Reallocation doubles size
  - **`vector<`*`type`*`> v(a, a + SIZE)`**
    - Creates **`vector v`** with elements from array **`a`** up to (not including) **`a + SIZE`**

21.2.1 vector Sequence Container

- **`vector`** functions
  - **`v.insert(`***iterator***`,`** *value* **`)`**
    - Inserts *value* before location of *iterator*
  - **`v.insert(`***iterator***`,`** *array* **`,`** *array + SIZE***`)`**
    - Inserts array elements (up to, but not including *array + SIZE)* into vector
  - **`v.erase( iterator )`**
    - Remove element from container
  - **`v.erase( iter1, iter2 )`**
    - Remove elements starting from **`iter1`** and up to (not including) **`iter2`**
  - **`v.clear()`**
    - Erases entire container

21.2.1 vector Sequence Container

- **`vector`** functions operations
  - **`v.front(), v.back()`**
    - Return first and last element
  - **`v.[elementNumber] = value;`**
    - Assign **`value`** to an element
  - **`v.at[elementNumber] = value;`**
    - As above, with range checking
    - **`out_of_bounds`** exception

21.2.1 vector Sequence Container

- **`ostream_iterator`**
  - **`std::ostream_iterator< `*`type`*` >`**
    ***`Name`*`( outputStream, separator );`**
    - ***`type`***: outputs values of a certain type
    - **`outputStream`**: iterator output location
    - **`separator`**: character separating outputs
- Example
  - **`std::ostream_iterator< int > output( cout, " " );`**
  - **`std::copy( iterator1, iterator2, output );`**
    - Copies elements from **`iterator1`** up to (not including) **`iterator2`** to output, an **`ostream_iterator`**

```
// Fig. 21.14: fig21_14.cpp
// Demonstrating standard library vector class template.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <vector>  // vector class-template definition

// prototype for function template printVector
template < class T >
void printVector( const std::vector< T > &integers2 );

int main()
{
   const int SIZE = 6;
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };

   std::vector< int > integers;

   cout << "The initial size of integers is: "
        << integers.size()
        << "\nThe initial capacity of integers is: "
        << integers.capacity();

```

Create a **`vector`** of **`int`**s.

Call member functions.

fig21_14.cpp
(1 of 3)

```
   // function push_back is in every sequence collection
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );

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

   cout << "\n\nOutput array using pointer notation: ";

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

   cout << "\nOutput vector using iterator notation: ";
   printVector( integers );

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

```

Add elements to end of **vector** using **push_back**.

fig21_14.cpp
(2 of 3)

```
   std::vector< int >::reverse_iterator reverseIterator;

   for ( reverseIterator = integers.rbegin();
         reverseIterator!= integers.rend();
         ++reverseIterator )
      cout << *reverseIterator << ' ';

   cout << endl;

   return 0;

} // end main

// function template for outputting vector elements
template < class T >
void printVector( const std::vector< T > &integers2 )
{
   std::vector< T >::const_iterator constIterator;

   for ( constIterator = integers2.begin();
         constIterator != integers2.end();
         constIterator++ )
      cout << *constIterator << ' ';

} // end function printVector
```

Walk through **vector** backwards using a **reverse_iterator**.

Template function to walk through **vector** forwards.

Outline

```
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
```

fig21_14.cpp
output (1 of 1)

21.2.2 list Sequence Container

- **`list`** container
  - Header **`<list>`**
  - Efficient insertion/deletion anywhere in container
  - Doubly-linked list (two pointers per node)
  - Bidirectional iterators
  - **`std::list<`** ***`type`*** **`>`** ***`name`*** **`;`**

21.2.2 list Sequence Container

- **`list`** functions for object **`t`**
  - **`t.sort()`**
    - Sorts in ascending order
  - **`t.splice(iterator, otherObject );`**
    - Inserts values from **`otherObject`** before **`iterator`**
  - **`t.merge( otherObject )`**
    - Removes **`otherObject`** and inserts it into **`t`**, sorted
  - **`t.unique()`**
    - Removes duplicate elements

21.2.2 list Sequence Container

- **`list`** functions
  - **`t.swap(otherObject);`**
    - Exchange contents
  - **`t.assign(iterator1, iterator2)`**
    - Replaces contents with elements in range of iterators
  - **`t.remove(value)`**
    - Erases all instances of **`value`**

Outline

fig21_17.cpp
(1 of 5)

```
// Fig. 21.17: fig21_17.cpp
// Standard library list class template test program.
#include <iostream>

using std::cout;
using std::endl;

#include <list>       // list class-template definition
#include <algorithm>  // copy algorithm

// prototype for function template printList
template < class T >
void printList( const std::list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };

   std::list< int > values;
   std::list< int > otherValues;

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

Create two **list** objects.

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

   values.sort();  // sort values

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

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(),
      array, array + SIZE );

   cout << "\nAfter insert, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert at end of values
   values.splice( values.end(), otherValues );

   cout << "\nAfter splice, values contains: ";
   printList( values );

   values.sort();  // sort values

   cout << "\nAfter sort, values contains: ";
   printList( values );

```

Various `list` member functions.

fig21_17.cpp
(2 of 5)

Outline

fig21_17.cpp
(3 of 5)

```
   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(),
      array, array + SIZE );
   otherValues.sort();

   cout << "\nAfter insert, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert into values
   // in sorted order
   values.merge( otherValues );

   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front();  // remove element from front
   values.pop_back();   // remove element from back

   cout << "\nAfter pop_front and pop_back:"
        << "\n   values contains: ";
   printList( values );

   values.unique();  // remove duplicate elements

   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.begin(), otherValues.end() );

   cout << "\nAfter assign, values contains: ";
   printList( values );

   // remove otherValues elements and insert into values
   // in sorted order
   values.merge( otherValues );

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

   values.remove( 4 );  // remove all 4s

   cout << "\nAfter remove( 4 ), values contains: ";
   printList( values );
```

```cpp

   cout << endl;

   return 0;

} // end main

// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template < class T >
void printList( const std::list< T > &listRef )
{
   if ( listRef.empty() )
      cout << "List is empty";

   else {
      std::ostream_iterator< T > output( cout, " " );
      std::copy( listRef.begin(), listRef.end(), output );

   } // end else

} // end function printList
```

fig21_17.cpp
(5 of 5)

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

fig21_17.cpp
output (1 of 1)

21.2.3 deque Sequence Container

- **deque** ("deek"): double-ended queue
  - Header **<deque>**
  - Indexed access using **[]**
  - Efficient insertion/deletion in front and back
  - Non-contiguous memory: has "smarter" iterators
- Same basic operations as **vector**
  - Also has
    - **push_front** (insert at front of **deque**)
    - **pop_front** (delete from front)

```
// Fig. 21.18: fig21_18.cpp
// Standard library class deque test program.
#include <iostream>

using std::cout;
using std::endl;

#include <deque>      // deque class-template definition
#include <algorithm>  // copy algorithm

int main()
{
   std::deque< double > values;
   std::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 ( int i = 0; i < values.size(); ++i )
      cout << values[ i ] << ' ';

```

Create a **deque**, use member functions.

fig21_18.cpp
(1 of 2)

```
   values.pop_front();  // remove first element

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

   // use subscript operator to modify element at location 1
   values[ 1 ] = 5.4;

   cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
   std::copy( values.begin(), values.end(), output );

   cout << endl;

   return 0;

} // 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
```

fig21_18.cpp
(2 of 2)

fig21_18.cpp
output (1 of 1)

21.3 Associative Containers

- Associative containers
  - Direct access to store/retrieve elements
  - Uses keys (search keys)
  - 4 types: **`multiset`**, **`set`**, **`multimap`** and **`map`**
    - Keys in sorted order
    - **`multiset`** and **`multimap`** allow duplicate keys
    - **`multimap`** and **`map`** have keys and associated values
    - **`multiset`** and **`set`** only have values

21.3.1 multiset Associative Container

- **`multiset`**
  - Header **`<set>`**
  - Fast storage, retrieval of keys (no values)
  - Allows duplicates
  - Bidirectional iterators
- Ordering of elements
  - Done by comparator function object
    - Used when creating multiset
  - For integer multiset
    - **`less<int>`** comparator function object
    - **`multiset< int, std::less<int> > myObject;`**
    - Elements will be sorted in ascending order

21.3.1 multiset Associative Container

- Multiset functions
  - **`ms.insert(`*`value`*`)`**
    - Inserts value into multiset
  - **`ms.count(`*`value`*`)`**
    - Returns number of occurrences of ***`value`***
  - **`ms.find(`*`value`*`)`**
    - Returns iterator to first instance of ***`value`***
  - **`ms.lower_bound(`*`value`*`)`**
    - Returns iterator to first location of ***`value`***
  - **`ms.upper_bound(`*`value`*`)`**
    - Returns iterator to location after last occurrence of ***`value`***

21.3.1 multiset Associative Container

- Class **`pair`**
  - Manipulate pairs of values
  - **`Pair`** objects contain **`first`** and **`second`**
    - **`const_iterators`**
  - For a **`pair`** object **`q`**

    `q = ms.equal_range(`*`value`*`)`
    - Sets **`first`** and **`second`** to **`lower_bound`** and **`upper_bound`** for a given ***`value`***

fig21_19.cpp
(1 of 3)

```
// Fig. 21.19: fig21_19.cpp
// Testing Standard Library class multiset
#include <iostream>

using std::cout;
using std::endl;

#include <set>  // multiset class-template definition

// define short name for multiset type used in this progra
typedef std::multiset< int, std::less< int > > ims;

#include <algorithm>  // copy algorithm

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };

   ims intMultiset;  // ims is typedef for "integer multiset"
   std::ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
        << " values of 15 in the multiset\n";

```

`typedef`s help clarify program. This declares an integer multiset that stores values in ascending order.

```
   intMultiset.insert( 15 );  // insert 15 in intMultiset
   intMultiset.insert( 15 );  // insert 15 in intMultiset

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

   // iterator that cannot be used to change element values
   ims::const_iterator result;

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

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

   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );

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

   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );

   cout << "\nAfter insert, intMultiset contains:\n";
   std::copy( intMultiset.begin(), intMultiset.end(), output );

```

Use member function **`find`**.

Outline

fig21_19.cpp
(3 of 3)

```
   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: "
        << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: "
        << *( intMultiset.upper_bound( 22 ) );

   // p represents pair of const_iterators
   std::pair< ims::const_iterator, ims::const_ite

   // use equal_range to determine lower and uppe
   // of 22 in intMultiset
   p = intMultiset.equal_range( 22 );

   cout << "\n\nequal_range of 22:"
        << "\n   Lower bound: " << *( p.first )
        << "\n   Upper bound: " << *( p.second );

   cout << endl;

   return 0;

} // end main
```

Use a **pair** object to get the lower and upper bound for **22**.

```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

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

fig21_19.cpp
output (1 of 1)

21.3.2 set Associative Container

- **`set`**
  - Header **`<set>`**
  - Implementation identical to **`multiset`**
  - Unique keys
    - Duplicates ignored and not inserted
  - Supports bidirectional iterators (but not random access)
  - **`std::set<`** ***`type`*****`, std::less<`*****`type`*****`> >`** ***`name`*****`;`**

21.3.3 multimap Associative Container

- **`multimap`**
  - Header **`<map>`**
  - Fast storage and retrieval of keys and associated values
    - Has key/value pairs
  - Duplicate keys allowed (multiple values for a single key)
    - One-to-many relationship
    - I.e., one student can take many courses
  - Insert **`pair`** objects (with a key and value)
  - Bidirectional iterators

21.3.3 multimap Associative Container

- Example

```
std::multimap< int, double, std::less< int > > mmapObject;
```

  – Key type **`int`**
  – Value type **`double`**
  – Sorted in ascending order
    - Use **`typedef`** to simplify code

```
typedef std::multimap<int, double, std::less<int>> mmid;
mmid mmapObject;
mmapObject.insert( mmid::value_type( 1, 3.4 ) );
```

  – Inserts key **`1`** with value **`3.4`**
  `- mmid::value_type` creates a **`pair`** object

fig21_21.cpp

```
// Fig. 21.21: fig21_21.cpp
// Standard library class multimap test program.
#include <iostream>

using std::cout;
using std::endl;

#include <map>  // map class-template definition

// define short name for multimap type used in this program
typedef std::multimap< int, double, std::less< int > > mmid;

int main()
{
   mmid pairs;

   cout << "There are currently " << pairs.count( 15 )
        << " pairs with key 15 in the multimap\n";

   // insert two value_type objects in pairs
   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\n";
```

Definition for a **`multimap`** that maps integer keys to double values.

Create multimap and insert key-value pairs.

```

   // insert five value_type objects in pairs
   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\

   // use const_iterator to walk through elements of pairs
   for ( mmid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   cout << endl;

   return 0;

} // end main
```

Use iterator to print entire **multimap**.

fig21_21.cpp
(2 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
```

Outline

fig21_21.cpp
output (1 of 1)

21.3.4 map Associative Container

- **`map`**
  - Header **`<map>`**
  - Like **`multimap`**, but only unique key/value pairs
    - One-to-one mapping (duplicates ignored)
  - Use **`[]`** to access values
  - Example: for **`map`** object **`m`**

    ```
    m[30] = 4000.21;
    ```

    - Sets the value of key 30 to **`4000.21`**
  - If subscript not in **`map`**, creates new key/value pair
- Type declaration
  - `std::map< int, double, std::less< int > >;`

fig21_22.cpp
(1 of 2)

```cpp
// Fig. 21.22: fig21_22.cpp
// Standard library class map test program.
#include <iostream>

using std::cout;
using std::endl;

#include <map>  // map class-template defi

// define short name for map type used in this program
typedef std::map< int, double, std::less< int > > mid;

int main()
{
   mid pairs;

   // insert eight value_type objects in 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

```

Again, use **`typedef`**s to simplify declaration.

Duplicate keys ignored.

```
   cout << "pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( mid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   // use subscript operator to change value for key 25
   pairs[ 25 ] = 9999.99;

   // use subscript operator insert value for key 40
   pairs[ 40 ] = 8765.43;

   cout << "\nAfter subscript operations, pairs contains:"
        << "\nKey\tValue\n";

   for ( mid::const_iterator iter2 = pairs.begin();
         iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t'
           << iter2->second << '\n';

   cout << endl;

   return 0;

} // end main
```

Can use subscript operator to add or change key-value pairs.

Outline

fig21_22.cpp
(2 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
```

Outline

fig21_22.cpp
output (1 of 1)

21.4 Container Adapters

- Container adapters
  - **`stack`**, **`queue`** and **`priority_queue`**
  - Not first class containers
    - Do not support iterators
    - Do not provide actual data structure
  - Programmer can select implementation
  - Member functions **`push`** and **`pop`**

21.4.1 stack Adapter

- **`stack`**
  - Header **`<stack>`**
  - Insertions and deletions at one end
  - Last-in, first-out (LIFO) data structure
  - Can use **`vector`**, **`list`**, or **`deque`** (default)
  - Declarations

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // default deque
```

- **`vector`**, **`list`**
  - Implementation of **`stack`** (default **`deque`**)
  - Does not change behavior, just performance (**`deque`** and **`vector`** fastest)

Outline

fig21_23.cpp
(1 of 3)

```
// Fig. 21.23: fig21_23.cpp
// Standard library adapter stack test program.
#include <iostream>

using std::cout;
using std::endl;

#include <stack>   // stack adapter definition
#include <vector>  // vector class-template definition
#include <list>    // list class-template definition

// popElements function-template prototype
template< class T >
void popElements( T &stackRef );

int main()
{
   // stack with default underlying deque
   std::stack< int > intDequeStack;

   // stack with underlying vector
   std::stack< int, std::vector< int > > intVectorStack;

   // stack with underlying list
   std::stack< int, std::list< int > > intListStack;

```

Create stacks with various implementations.

Outline

21_23.cpp
(2 of 3)

```
   // push the values 0-9 onto each stack
   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );

   } // end for

   // 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;

   return 0;

} // end main

```

Use member function **push**.

```cpp
// pop elements from stack object to which stackRef refers
template< class T >
void popElements( T &stackRef )
{
   while ( !stackRef.empty() ) {
      cout << stackRef.top() << ' ';  // view top element
      stackRef.pop();                 // remove top element

   } // end while

} // end function popElements
```

```
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
```

Outline

fig21_23.cpp
(3 of 3)

fig21_23.cpp
output (1 of 1)

21.4.2 queue Adapter

- **`queue`**
  - Header **`<queue>`**
  - Insertions at back, deletions at front
  - First-in-first-out (FIFO) data structure
  - Implemented with **`list`** or **`deque`** (default)
    - **`std::queue<double> values;`**
- Functions
  - **`push( element )`**
    - Same as **`push_back`**, add to end
  - **`pop( element )`**
    - Implemented with **`pop_front`**, remove from front
  - **`empty()`**
  - **`size()`**

fig21_24.cpp
(1 of 2)

```
// Fig. 21.24: fig21_24.cpp
// Standard library adapter queue test program.
#include <iostream>

using std::cout;
using std::endl;

#include <queue>  // queue adapter definition

int main()
{
   std::queue< double > values;

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

   cout << "Popping from values: ";

   while ( !values.empty() ) {
      cout << values.front() << ' ';  // view front element
      values.pop();                   // remove element

   } // end while

```

Create **queue**, add values using **push**.

Outline

```
   cout << endl;

   return 0;

} // end main
```

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

fig21_24.cpp
(2 of 2)

fig21_24.cpp
output (1 of 1)

21.5 Algorithms

- Before STL
  - Class libraries incompatible among vendors
  - Algorithms built into container classes
- STL separates containers and algorithms
  - Easier to add new algorithms
  - More efficient, avoids **`virtual`** function calls
  - **`<algorithm>`**

21.5.6 Basic Searching and Sorting Algorithms

- **`find(iter1, iter2, value)`**
  - Returns iterator to first instance of **`value`** (in range)
- **`find_if(iter1, iter2, function)`**
  - Like **`find`**
  - Returns iterator when **`function`** returns **`true`**
- **`sort(iter1, iter2)`**
  - Sorts elements in ascending order
- **`binary_search(iter1, iter2, value)`**
  - Searches ascending sorted list for value
  - Uses binary search

```
// Fig. 21.31: fig21_31.cpp
// Standard library search and sort algorithms.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>  // algorithm definitions
#include <vector>     // vector class-template definition

bool greater10( int value );  // prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };

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

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

   // locate first occurrence of 16 in v
   std::vector< int >::iterator location;
   location = std::find( v.begin(), v.end(), 16 );
```

fig21_31.cpp
(1 of 4)

Outline

fig21_31.cpp
(2 of 4)

```cpp

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

   // locate first occurrence of 100 in v
   location = std::find( v.begin(), v.end(), 100 );

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

   // locate first occurrence of value greater than 10 in v
   location = std::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";

```

fig21_31.cpp
(3 of 4)

```
   // sort elements of v
   std::sort( v.begin(), v.end() );

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

   // use binary_search to locate 13 in v
   if ( std::binary_search( v.begin(), v.end(), 13 ) )
      cout << "\n\n13 was found in v";
   else
      cout << "\n\n13 was not found in v";

   // use binary_search to locate 100 in v
   if ( std::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;

} // end main

```

Outline

fig21_31.cpp
(4 of 4)

```
// determine whether argument is greater than 10
bool greater10( int value )
{
   return value > 10;

} // end function greater10
```

fig21_31.cpp
output (1 of 1)

```
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
```

21.7 Function Objects

- Function objects (**`<functional>`**)
  - Contain functions invoked using operator **`()`**

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

fig21_42.cpp
(1 of 4)

```
// Fig. 21.42: fig21_42.cpp
// Demonstrating function objects.
#include <iostream>

using std::cout;
using std::endl;

#include <vector>      // vector class-template definition
#include <algorithm>   // copy algorithm
#include <numeric>     // accumulate algorithm
#include <functional>  // binary_function definition

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

} // end function sumSquares

```

Create a function to be used with **`accumulate`**.

fig21_42.cpp

```
// binary function class template defines overloaded operator()
// that adds suare of its second argument and running total in
// its first argument, then returns sum
template< class T >
class SumSquaresClass : public std::binary_function< T, T, T > {

public:

   // add square of value to total and return result
   const T operator()( const T &total, const T &value )
   {
      return total + value * value;

   } // end function operator()

}; // end class SumSquaresClass

```

Create a function object (it can also encapsulate data). Overload **operator()**.

```
int main()
{
   const int SIZE = 10;
   int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   std::vector< int > integers( array, array + SIZE );

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

   int result = 0;

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

   // calculate sum of squares of elements of vector integers
   // using binary function sumSquares
   result = std::accumulate( integers.begin(), integers.end(),
      0, sumSquares );

   cout << "\n\nSum of squares of elements in integers using "
        << "binary\nfunction sumSquares: " << result;

```

`accumulate` initially passes **`0`** as the first argument, with the first element as the second. It then uses the return value as the first argument, and iterates through the other elements.

fig21_42.cpp
(3 of 4)

Outline

fig21_42.cpp
(4 of 4)

fig21_42.cpp
output (1 of 1)

```
   // calculate sum of squares of elements of vector integers
   // using binary-function object
   result = std::accumulate( integers.begin(), integers.end(),
      0, SumSquaresClass< int >() );

   cout << "\n\nSum of squares of elements in integers using "
        << "binary\nfunction object of type "
        << "SumSquaresClass< int >: " << result << endl;

   return 0;

} // end main
```

Use **accumulate** with a function object.

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


Sum of squares of elements in integers using binary
function sumSquares: 385


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