Standard Template Library (STL)

Computer Programming for Engineers
(DASF003-41)

Instructor:
Sungjae Hwang (jason.sungjae.hwang@gmail.com)

Today

- **Introduction**
  - by Example: Vector Template Class
  - Introduction to STL
- **Iterators**
  - Constant, mutable, and reverse iterators
- **Containers and adaptors**
  - Sequential containers: vector
  - Associative Containers: set and map

INTRODUCTION BY EXAMPLE: VECTOR TEMPLATE CLASS

Introduction to Vectors

- **Limitation of C-Arrays**
  - The size of static array is fixed, and should be known at compile time
  - Dynamic array needs malloc/free/new/delete, which needs to be handled with care.

- **STL Vectors: "arrays that automatically grow and shrink"**
  - Array-like data structure dynamically resized during program execution
  - However, we do not care about the memory allocation/deallocation.

- **Declared differently:**
  - Syntax:
    - `std::vector<Base_Type>`
    - Produces "new" class for vectors with that type
  - Example declaration:
    - `std::vector<int> v;`

Vector Usage

```
std::vector<int> v;
```

- "v is vector of type int"
- Calls class default constructor: empty vector object created

Usage

- Indexing: indexed like arrays for access (e.g., v[0], v[1], v[k],...)
- Adding elements: `push_back()`
- Querying the count of elements: `size()`
- Many other convenient member functions
  - https://www.cplusplus.com/reference/vector/vector/

Vector Example

Display 7.7 Using a Vector (1 of 2)

```
#include <iostream>
#include <vector>
using namespace std;
int main( )
{
   vector<int> v;
   cout << "Enter a list of positive numbers.\n"
      << "Place a negative number at the end.\n";
   int next;
   cin >> next;
   while( next > 0)
   {
      v.push_back(next);
      cout << next << " added. ";
      cout << "v.size( ) = " << v.size( ) << endl;
      cin >> next;
   }
```

Vector Example

Display 7.7 Using a Vector (1 of 2)

```
    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
       cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8
```

Vector Efficiency

Member function capacity()

- Returns memory currently allocated
- Not same as size()
- Typically, capacity >= size
  - Automatically increased as needed
  - In practice, when capacity is not enough, the capacity is **doubled**.

If efficiency critical:

- Can set behaviors manually

```
v.reserve(32);           // pre-set capacity to 32
v.reserve(v.size()+10); // allocates 10 more elements
```

Vector capacity

- see how capacity increases.

```
 1 #include <iostream>
 2 #include <vector>
 3 using namespace std;
 5 int main(){
 7   vector<int> v;
 8   cout << "capacity: " << v.capacity() << endl;
 9   int num[] = {0,1,2,3,4,5};
10
11   for(auto i : num) {
12     v.push_back(i);
13     cout << "after insert " << i << ", capacity: " << v.capacity() << endl;
14   }
15
16   for(auto i = 0; i < v.size(); i++) {
17     cout << v[i] << endl;
18   }
19
20   cout << "capacity: " << v.capacity() << endl;
21   v.reserve(100);
22   cout << "capacity: " << v.capacity() << endl;
24   return 0;
25 }
```

STANDARD TEMPLATE LIBRARY (STL)

Introduction

Standard Template Library (STL)

- Set of C++ template classes
- Software library for C++, having all such data structures
- Code quickly, efficiency, generic programming

Main components

- Container
- Iterator
- Algorithm
- Adaptors

STL Components

Container

- Stores objects or data of arbitrary types

Iterator

- Step through elements in containers

Algorithm

- Performs particular tasks using iterator
- Sort, search

Adaptors

- Wrapping common container to implement data structures
- deque : stack and queue
- Vector : priority_queue

Standard Containers in STL

Sequence containers: ordered collections

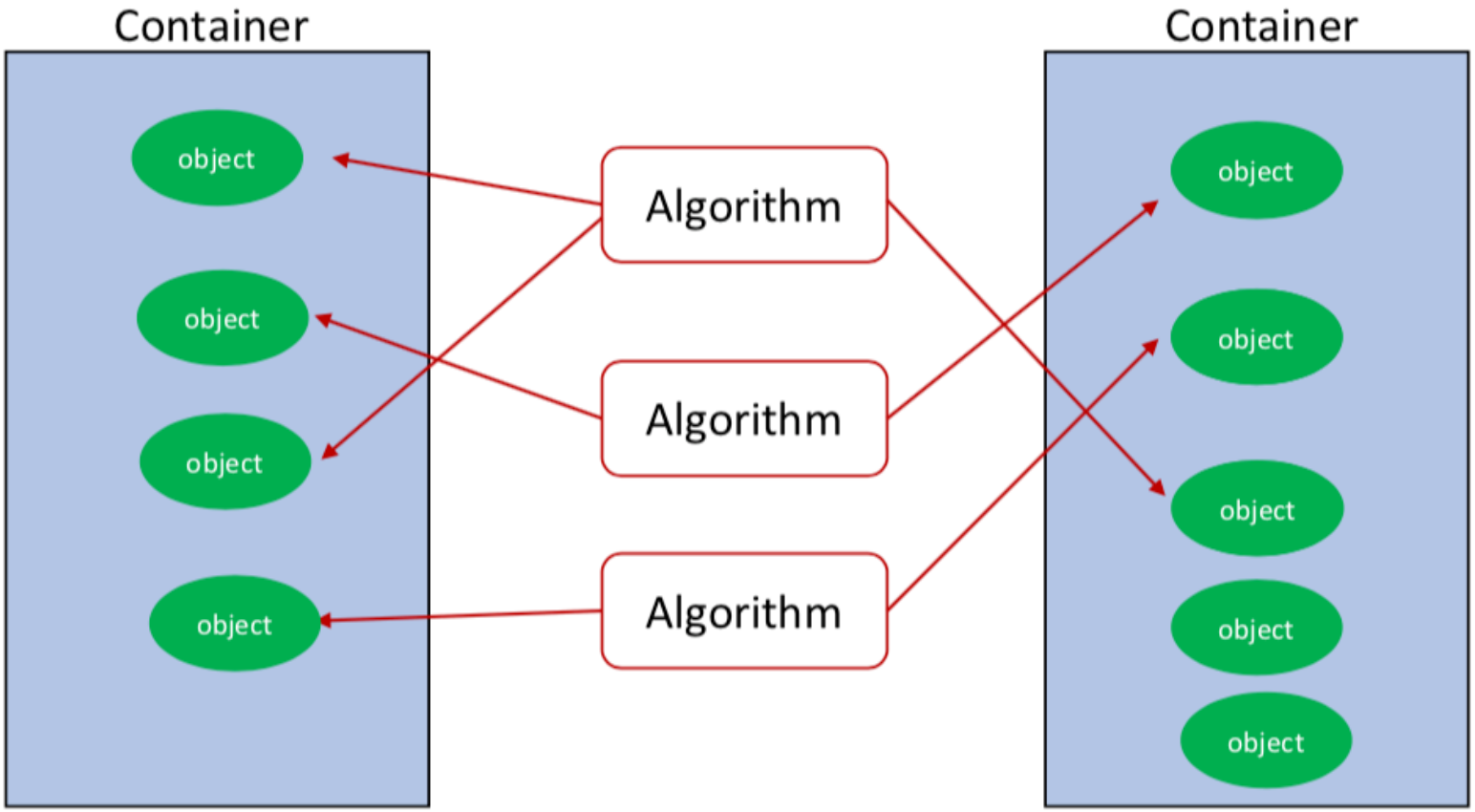
- `vector`: dynamic array
- `list`: doubly linked list
- `deque`: double-ended queue (adapted to stack and queue)

Associative containers: unordered collections

- `set`, `multiset`
- `map`: dictionary (internally ordered by balanced binary tree)
- `multimap`: similar to map, but with duplicate keys
- `unordered_map`: dictionary with hash

Container, Iterators, Algorithms

- Algorithm uses iterators to access the objects in containers



ITERATORS

Iterators

- **Generalization of a pointer to STL containers/adaptors**
  - Typically even implemented with pointer!
- **"Abstraction" of iterators**
  - Designed to hide details of implementation
  - Provide consistent interface across different container classes
- **Each container class has "own" iterator type**
  - Similar to how each data type has own pointer type

Manipulating Iterators

Recall using overloaded operators:

- `++, --, ==, !=, *`
- So if `p` is an iterator variable, `*p` gives access to data pointed to by `p`

Vector template class

- Has all above overloads
- Also has members `begin()` and `end()`

```
// return iterator for the first item in c
std::vector<int>::iterator it = c.begin();
// return iterator for after-last item in c
// e.g., for size-2 vector, end() indicates index 2
auto it2 = c.end();
```

Cycling with Iterators

Recall cycling ability:

- Using `begin()/end()`, we can write for-loop in a similar way used for arrays

```
for( auto p = c.begin(); p != c.end(); p++ )
   process(*p); //*p is current data item
```

- Powerful usage of `auto`!

Keep in mind:

- Each container type in STL has own iterator types
- Even though they're all used similarly

Vector Cycling Example

Display 19.1 Iterators Used with a Vector (1 of 2)

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;     // Using only a part of std
using std::endl;
using std::vector;

int main( )
{
   vector<int> container;
   for (int i = 1; i <= 4; i++)  container.push_back(i);
   cout << "Here is what is in the container:\n";
   vector<int>::iterator p;
```

Vector Cycling Example

Display 19.1 Iterators Used with a Vector (2 of 2)

```
    for (p = container.begin( ); p != container.end( ); p++)
       cout << *p << " ";
    cout << endl;
    cout << "Setting entries to 0:\n";
    for (p = container.begin( ); p != container.end( ); p++)
       *p = 0;
    cout << "Container now contains:\n";
    for (p = container.begin( ); p != container.end( ); p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

```
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
```

Vector Iterator Types

- Iterators for vectors of integers are of type:

```
std::vector<int>::iterator
```

- Iterators for lists of integers are of type:

```
std::list<int>::iterator
```

Iterator Classifications

- **Forward iterators:**
  - ++ works on iterator
- **Bidirectional iterators:**
  - Both ++ and -- work on iterator
- **Random-access iterators:**
  - ++, --, and random access all work with iterator

- **These are kinds of “iterators”, not types!**

Random Access

Display 19.2

- Bidirectional and Random-Access Iterator Use (1 of 3)

```
int main()
{
   vector<char> container;
   container.push_back('A');
   container.push_back('B');
   container.push_back('C');
   container.push_back('D');
   for (int i = 0; i < 4; i++)
      cout << "container[" << i << "] == "
         << container[i] << endl;
   vector<char>::iterator p = container.begin();

   cout << "The third entry is " << container[2] << endl;
   cout << "The third entry is " << p[2] << endl;
   cout << "The third entry is " << *(p + 2) << endl;
```

Three different notations for the same thing.

Random Access

Display 19.2

- Bidirectional and Random-Access Iterator Use (2 of 3)

```
    cout << "Back to container[0].\n";
    p = container.begin( );
    cout << "which has value " << *p << endl;
    cout << "Two steps forward and one step back:\n";
    p++;
    cout << *p << endl;
    p++;
    cout << *p << endl;
    p--;
    cout << *p << endl;
    return 0;
}
```

```
p++ moves the iterator.
So, p[2] will show
different
Results.
```

Random Access

Display 19.2

- Bidirectional and Random-Access Iterator Use (3 of 3)

```
container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

Constant and Mutable Iterators

- **Dereferencing operator's behavior dictates**
- **Constant iterator:**
  - * produces read-only version of element
  - Can use *p to assign to variable or output, but cannot change element in container
    - `*p = <anything>; // is illegal`
- **Mutable iterator:**
  - *p can be assigned value
  - Changes corresponding element in container
  - i.e.: *p returns an lvalue

Reverse Iterators

To cycle elements in reverse order

- Requires container with *bidirectional* iterators
- Might consider:

```
for( auto p=container.end(); p!=container.begin(); p-- )
   cout << *p << " " ;
```

- But recall: `end()` is just "sentinel", `begin()` is not!

Use reverse iterators to cycle elements in reverse order:

```
for(vector<int>::reverse_iterator rp=c.rbegin();rp!=c.rend(); rp++)
   cout << *rp << " " ;
```

- `rbegin()`: returns iterator at last element
- `rend()`: returns sentinel "end" marker

Reverse Iterator


```
 1 #include <iostream>
 2 #include <vector>
 3 using namespace std;
 5 int main(){
 7   vector<char> container;
 8   container.push_back('A');
 9   container.push_back('B');
10   container.push_back('C');
11   container.push_back('D');
12
13   for (auto it = container.begin(); it != container.end(); it++)
14     cout << *it << " ";
15   cout << endl;
16
17   cout << "Print in a reverse order\n";
18   // What happens with the below line?
19   //for (auto it = container.end(); it != container.begin(); it--)
20   for (vector<char>::reverse_iterator it = container.rbegin(); it != container.rend();
it++)
21     cout << *it << " ";
22   cout << endl;
24   return 0;
26 }
```

CONTAINERS

Containers

- **Container classes in STL**
  - Different kinds of data structures
  - Linked lists, queues, stacks
- **Each with parameter for particular data type to be stored**
  - e.g., Lists of ints, doubles or myClass types
- **Each has own iterators**
  - One might have bidirectional, another might just have forward iterators
  - But all operators and members have same meaning

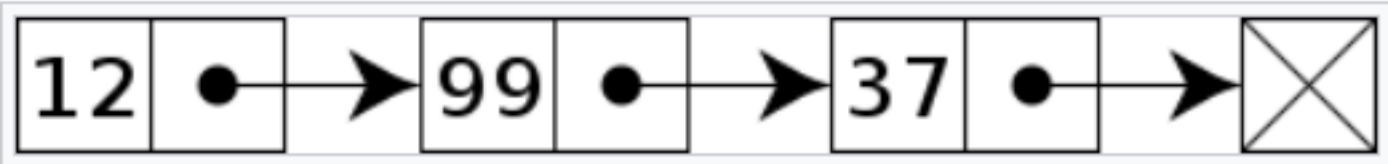
Sequential Containers

Arranges list data

- 1st element, next element, … to last element
- Vector is a container class

Linked list is sequential container

- Linear collection of data elements
- Each element points to the next element



- https://en.wikipedia.org/wiki/Linked_list

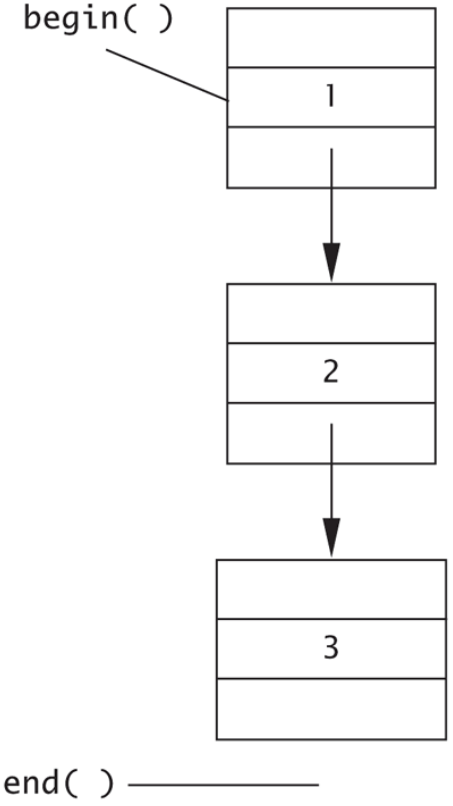
STL has no "singly linked list"

- Only "doubly linked list": template class *list*

Display 19.4 Two Kinds of Lists

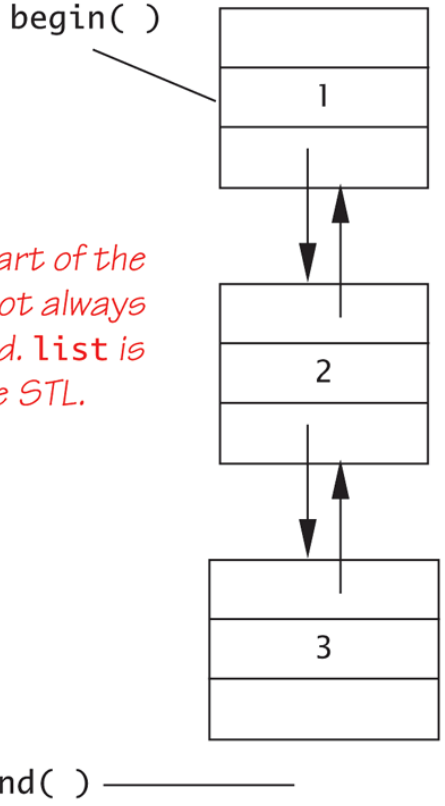
Display 19.4 Two Kinds of Lists

`slist`: *A singly linked list*
++ defined; -- not defined



`slist` *is not part of the STL and may not always be implemented.* `list` *is part of the STL.*

`list`: *A doubly linked list*
Both ++ and -- defined



list Template Class

Display 19.5 Using the list Template Class(1 of 2)

```
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main( )
{
   list<int> listObject;
   for (int i = 1; i <= 3; i++)
      listObject.push_back(i);

   cout << "List contains:\n";
   list<int>::iterator iter;
   for (iter = listObject.begin( ); iter != listObject.end( ); iter++)
      cout << *iter << " ";
   cout << endl;
```

list Template Class

Display 19.5 Using the list Template Class(2 of 2)

```
    cout << "Setting all entries to 0:\n";
    for (iter = listObject.begin( ); iter != listObject.end( ); iter++)
       *iter = 0;
    cout << "List now contains:\n";
    for (iter = listObject.begin( ); iter != listObject.end( ); iter++)
       cout << *iter << " ";
    cout << endl;
    return 0;
}
```

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

List

- Source code is shown in the prior slides.


```
// Random access is not defined.
//iter = listObject.begin();
//cout << iter[2] << endl; // Error
```

Associative Containers

- **Associative container:**
  - simple database or dictionary
  - Store data with key: each data item has key

- **Example:**
  - data: employee's record as struct
  - key: employee's SSN
  - Items retrieved based on key

`set` Template Class

- Simplest container possible
- Stores elements without repetition
  - 1st insertion places element in set
- Each element is own key

- Capabilities:
  - Add elements
  - Delete elements
  - Ask if element is in set

class template

std::set <set>

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

Set

Sets are containers that store unique elements following a specific order.

In a `set`, the value of an element also identifies it (the value is itself the *key*, of type `T`), and each value must be unique. The value of the elements in a `set` cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a `set` are always sorted following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`set` containers are generally slower than unordered_set containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as *binary search trees*.

More `set` Template Class

- **Designed to be efficient**
  - Stores values in sorted order
  - Can specify order: `set<T, Ordering> s;`
    - `i.e., set<int,greater<int>>, set<int,CustomOrder>`
    - Ordering is well-behaved ordering relation that returns bool
    - None specified: use < relational operator

`set` Template Class Example

Program Using the set Template Class (1 of 3)

```
//Program to demonstrate use of the set template class.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main( )
{
   set<char> s;
   s.insert('A');
   s.insert('D');
   s.insert('D');
   s.insert('C');
   s.insert('C');
   s.insert('B');
```

Set Template Class Example

Program Using the set Template Class (2 of 3)

```
cout << "The set contains:\n";
set<char>::const_iterator p;

for (p = s.begin( ); p != s.end( ); p++)
   cout << *p << " ";
cout << endl;

cout << "Set contains 'C': ";
if (s.find('C')==s.end( ))
   cout << " no " << endl;
else cout << " yes " << endl;

cout << "Removing C.\n";
s.erase('C');
```

Set Template Class Example

Program Using the set Template Class (3 of 3)

```
   for (p = s.begin( ); p != s.end( ); p++)
      cout << *p << " ";
   cout << endl;
   cout << "Set contains 'C': ";

   if (s.find('C')==s.end( ))
      cout << " no " << endl;
   else cout << " yes " << endl;
   return 0;
}
```

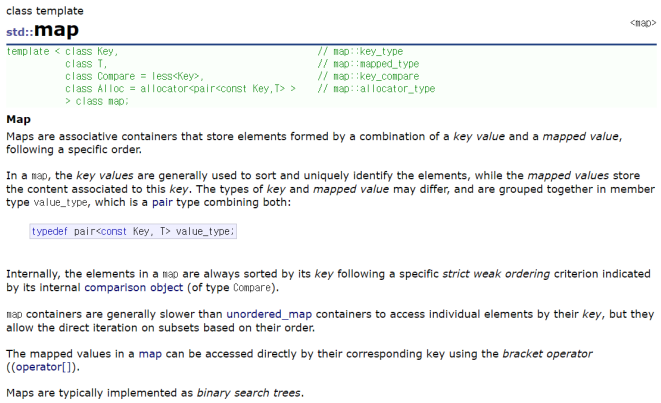
```
The set contains:
A B C D
Set contains 'C': yes
Removing C.
A B D
Set contains 'C': no
```

Map Template Class

- A function given as set of ordered pairs
  - For each value *first*, at most one value *second* in map considering (*first*, *second*) pair
- Example map declaration:

```
map<string, int> numberMap;
```

- Can use [ ] notation to access the map
  - For both storage and retrieval
- Stores in sorted order, like set
  - Second value can have no ordering impact



class template

std::**map** <map>

```
template < class Key,                                // map::key_type
           class T,                                  // map::mapped_type
           class Compare = less<Key>,                // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a `pair` type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal `comparison object` (of type `Compare`).

`map` containers are generally slower than `unordered_map` containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a `map` can be accessed directly by their corresponding key using the *bracket operator* (`operator[]`).

Maps are typically implemented as *binary search trees*.

Map Template Class Example

Program Using the map Template Class(1 of 3)

```
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main( )
{
   map<string, string> planets;
   planets["Mercury"] = "Hot planet";
   planets["Venus"] = "Atmosphere of sulfuric acid";
   planets["Earth"] = "Home";
   planets["Mars"] = "The Red Planet";
   planets["Jupiter"] = "Largest planet in our solar system";
```

Map Template Class Example

Program Using the map Template Class(2 of 3)

```
planets["Saturn"] = "Has rings";
planets["Uranus"] = "Tilts on its side";
planets["Neptune"] = "1500 mile-per-hour winds";
planets["Pluto"] = "Dwarf planet";

cout << "Entry for Mercury - " << planets["Mercury"]
   << endl << endl;

if (planets.find("Mercury") != planets.end( ))
   cout << "Mercury is in the map." << endl;
if (planets.find("Ceres") == planets.end( ))
   cout << "Ceres is not in the map." << endl << endl;

cout << "Iterating through all planets: " << endl;
```

Map Template Class Example

Program Using the map Template Class(2 of 3)

```
    map<string, string>::const_iterator iter;
    //The iterator will output the map in order sorted by the key.
    for (iter = planets.begin( ); iter != planets.end( ); iter++)
    {
        cout << iter->first << " - " << iter->second << endl;
    }
    return 0;
}
```

```
Entry for Mercury - Hot planet
Mercury is in the map.
Ceres is not in the map.
Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
…
```

Summary

- Vector template class
- Standard Template Library (STL)
- Iterator
- Container