CIT237

Chapter 17: The Standard Template Library

November 18, 2019

Reminders

- Quiz 6 will be held at the start of class on Wednesday, November 20.
- The material covered on Quiz 6 will be:
  - Lectures of October 28 through November 13.
  - Chapters 15 and 16.
- Project 3:

  Due date is December 2.

The Standard Template Library

- **The Standard Template Library (STL)**: an extensive library of generic templates for classes and functions.
- Categories of Templates:
  - **Containers**: Class templates for objects that store and organize data
  - **Iterators**: Class templates for objects that behave like pointers, and are used to access the individual data elements in a container
  - **Algorithms**: Function templates that perform various operations on elements of containers

Containers

- **Sequence Containers**
  - Stores data “sequentially”, in a fashion similar to an array: they appear (to the programmer) that they are sequential.
  - Some sequence containers *actually* use contiguous memory addresses, others do not.

- **Associative Containers**
  - Store data in a nonsequential way that can make it faster to locate particular elements in the container.

STL Header Files

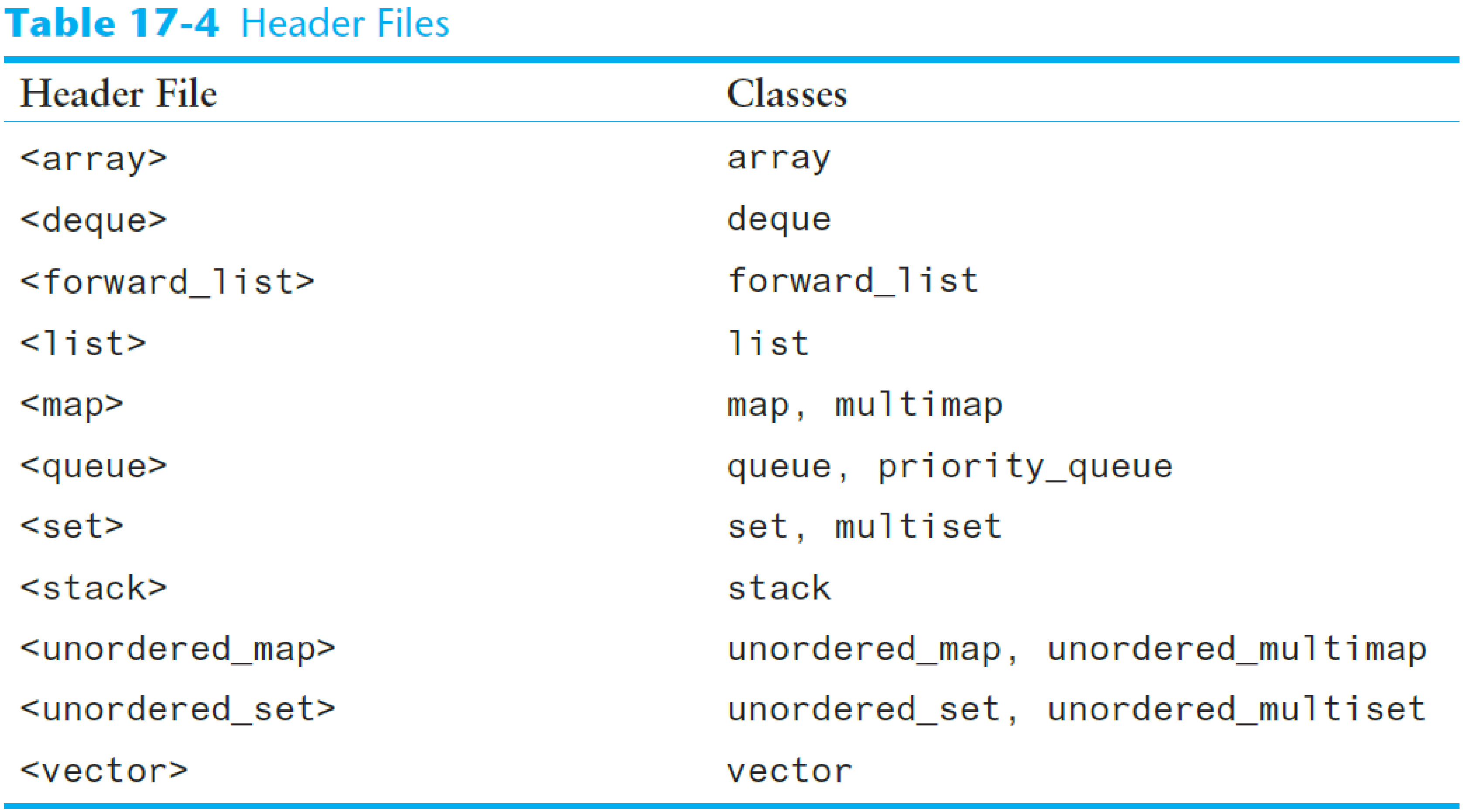


Table 17-4 Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

The `array` Class Template (1)

- An `array` object works very much like a regular array

- A fixed-size container that holds elements of the same data type.

- `array` objects have a **`size()`** member function that returns the number of elements contained in the object.

The `array` Class Template (2)

- The `array` class is declared in the `<array>` header file.

- When defining an `array` object, you specify the data type of its elements, and the number of elements.

- Examples:

```
array<int, 5> numbers;

array<string, 4> names;
```

Iterators

- Objects that work like pointers
- Used to access data in STL containers
- Five categories of iterators:

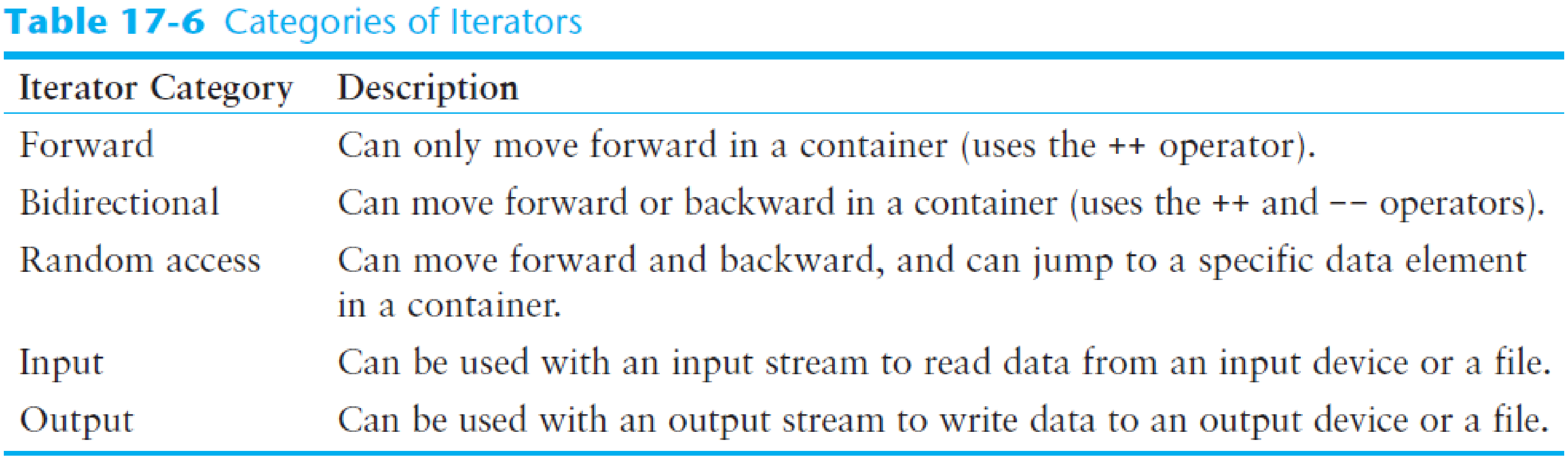


Table 17-6 Categories of Iterators

Iterator Category	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and -- operators).
Random access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Similarities between Pointers and Iterators

	Pointers	**Iterators**
Use the * and -> operators to dereference	Yes	Yes
Use the = operator to assign to an element	Yes	Yes
Use the == and != operators to compare	Yes	Yes
Use the ++ operator to increment	Yes	Yes
Use the -- operator to decrement	Yes	Yes (bidirectional and random-access iterators)
Use the + operator to move forward a specific number of elements	Yes	Yes
Use the - operator to move backward a specific number of elements	Yes	Yes (bidirectional and random-access iterators)

Defining an Iterator

- To define an iterator, you must know what type of container you will be using it with.

- The general format of an iterator definition:

  containerType`::iterator` *iteratorName*`;`

  Where *containerType* is the STL container type, and *iteratorName* is the name of the iterator variable that you are defining.

Iterator Examples (1)

- For example, suppose we have defined an `array` object, as follows:

  ```
  array<string, 3> names = {"Sarah", "William", "Alfredo"};
  ```

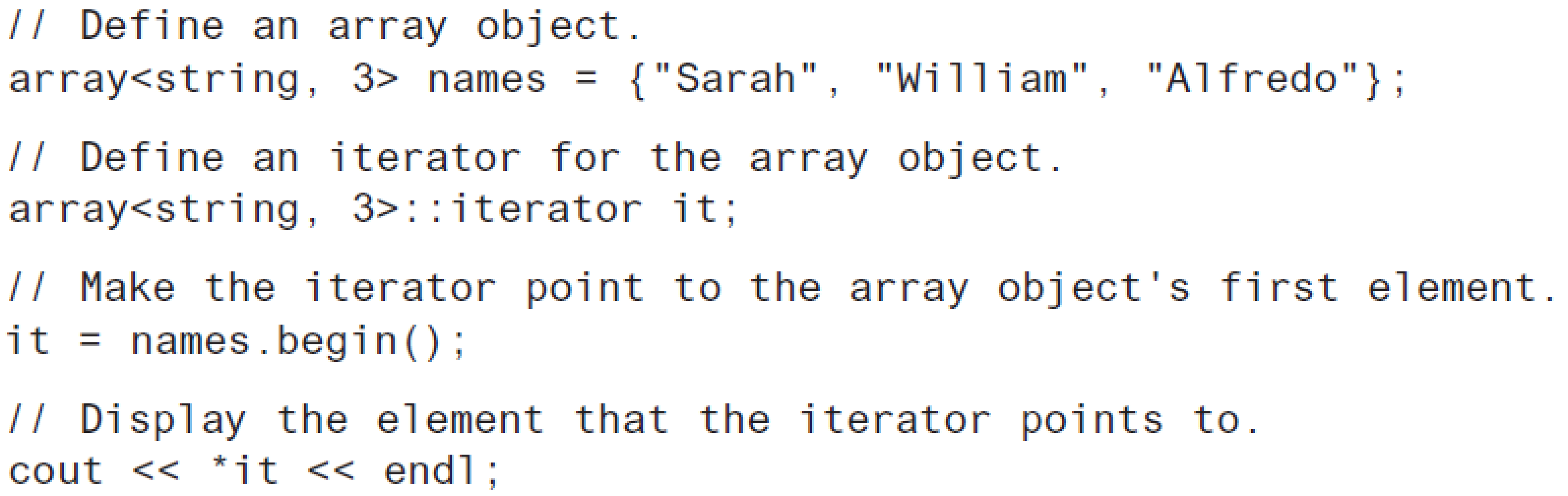
  We can define an iterator that is compatible with the array object as follows:

  ```
  array<string, 3>::iterator it;
  ```

  This defines an iterator named `it`. The iterator can be used with an `array<string, 3>` object.

Iterator Examples (2)

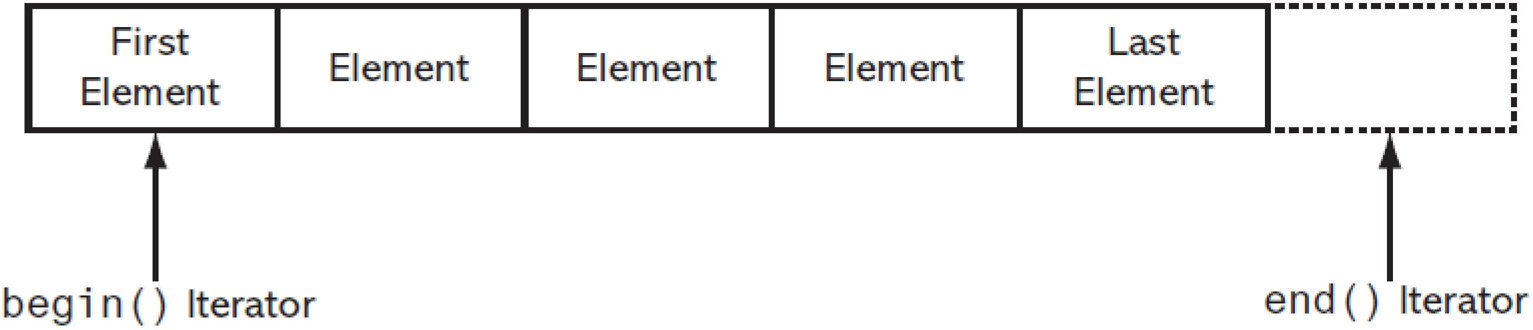
- All of the STL containers have a `begin()` member function that returns an iterator pointing to the container's first element.



```
// Define an array object.
array<string, 3> names = {"Sarah", "William", "Alfredo"};

// Define an iterator for the array object.
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element.
it = names.begin();

// Display the element that the iterator points to.
cout << *it << endl;
```

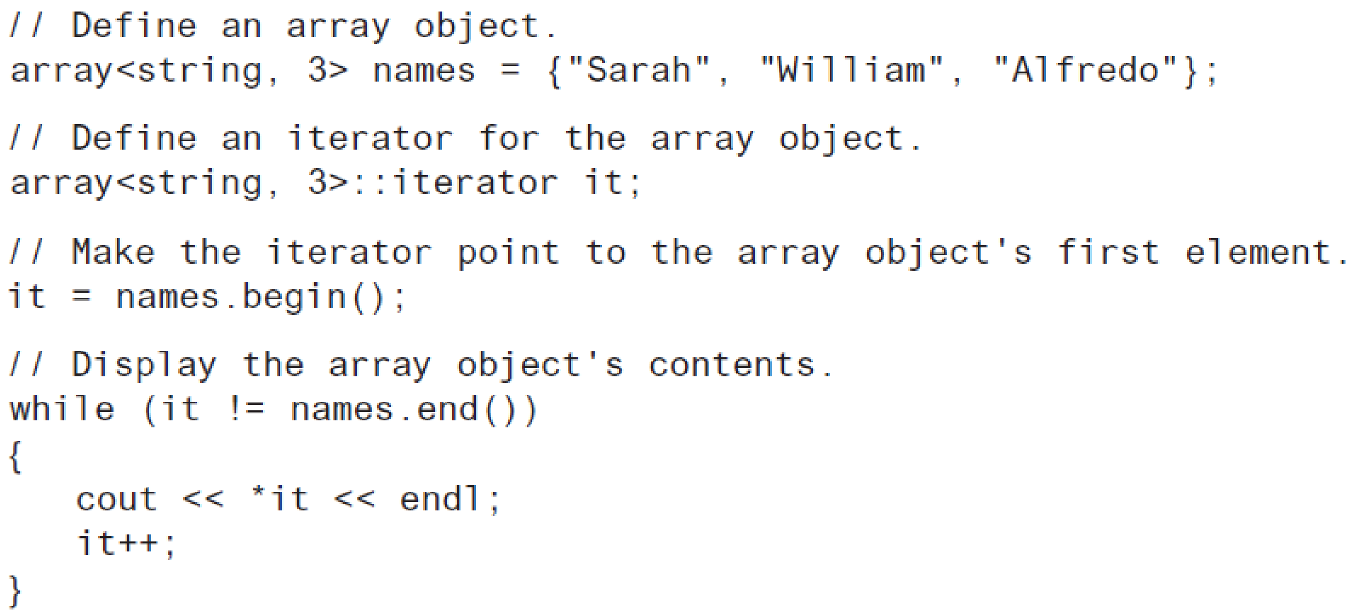
Iterator Examples (3)

- All of the STL containers have a `end()` member function that returns an iterator pointing to the position *after* the container's last element.



Iterator Examples (4)

- You typically use the `end()` member function to know when you have reached the end of a container.



```
// Define an array object.
array<string, 3> names = {"Sarah", "William", "Alfredo"};

// Define an iterator for the array object.
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element.
it = names.begin();

// Display the array object's contents.
while (it != names.end())
{
   cout << *it << endl;
   it++;
}
```

The `vector` Class

- A `vector` is a sequence container that works like an array, but is dynamic in size.
- Overloaded `[]` operator provides access to existing elements
- The `vector` class is declared in the `<vector>` header file.

Textbook "Vector-related" Sections

- Section 7.11: Vectors
  - pages 435-449
- Section 8.5: Sorting and Searching Vectors
  - pages 495-498
- Section 17.3: The `vector` Class
  - pages 1040 - 1053

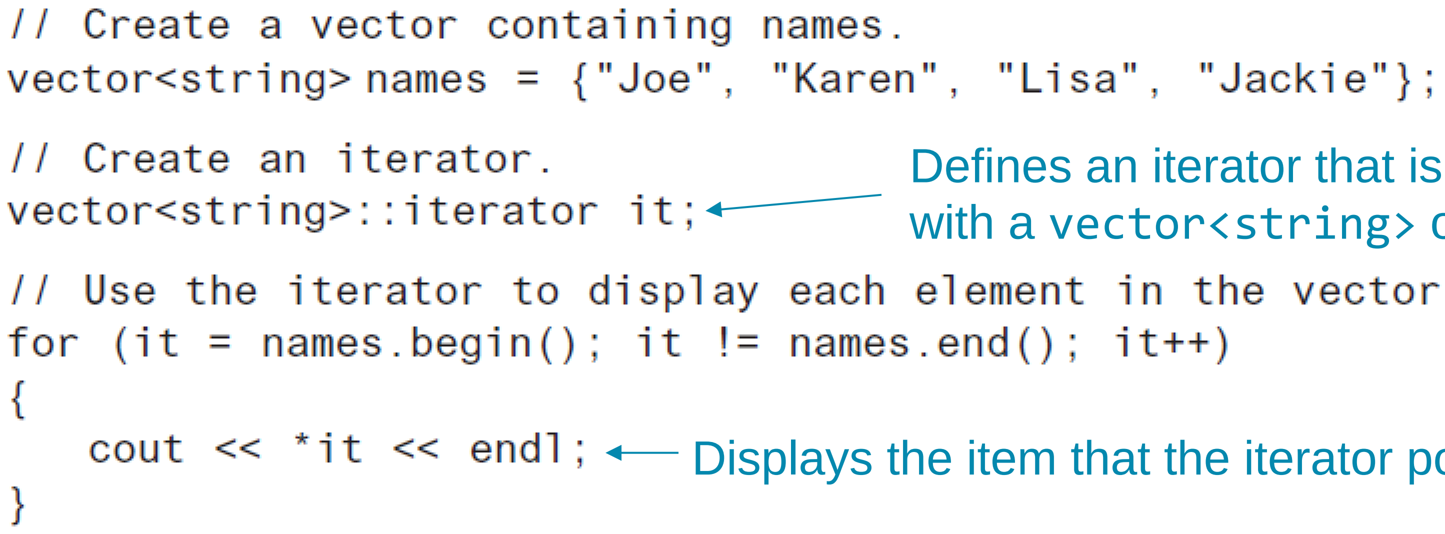
Adding New Elements to a `vector`

- The `push_back` member function adds a new element to the end of a `vector`:

```
vector<int> numbers;
numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
```

Using an Iterator With a `vector`

- `vector`s have `begin()` and `end()` member functions that return iterators pointing to the beginning and end of the container:



```
// Create a vector containing names.
vector<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
vector<string>::iterator it;

// Use the iterator to display each element in the vector.
for (it = names.begin(); it != names.end(); it++)
{
    cout << *it << endl;
}
```

Defines an iterator that is compatible with a `vector<string>` object

Displays the item that the iterator points to

Using an Iterator With a `vector`

- The `begin()` and `end()` member functions return a random-access iterator of the `iterator` type

- The `cbegin()` and `cend()` member functions return a random-access iterator of the `const_iterator` type

- The `rbegin()` and `rend()` member functions return a reverse iterator of the `reverse_iterator` type

- The `crbegin()` and `crend()` member functions return a reverse iterator of the `const_reverse_iterator` type

The **insert()** Member Function

- You can use the **insert()** member function, along with an iterator, to insert an element at a specific position.
- General format:

```
vectorName.insert(it, value);
```

Iterator pointing to an element in the vector

Value to insert before the element that *it* points to

Emplacement of Container Elements (1)

- Member functions such as **insert()** and **push_back()** can cause temporary objects to be created in memory while the insertion is taking place.
- C++11 introduced a new family of member functions that use a technique known as *emplacement* to insert new elements.
- Emplacement avoids the creation of temporary objects in memory while a new object is being inserted into a container.
- The emplacement functions are more efficient than functions such as insert() and push_back()

Emplacement of Container Elements (2)

- The `vector` class provides two member functions that use emplacement:
  - **`emplace()`** - emplaces an element at a specific location
  - **`emplace_back()`**- emplaces an element at the end of the vector

- With these member functions, it is not necessary to instantiate, ahead of time, the object you are going to insert.
- Instead, you pass to the emplacement function any arguments that you would normally pass to the constructor of the object you are inserting.
- The emplacement function handles the construction of the object, forwarding the arguments to its constructor.

Maps – General Concepts

- A *map* is an associative container.

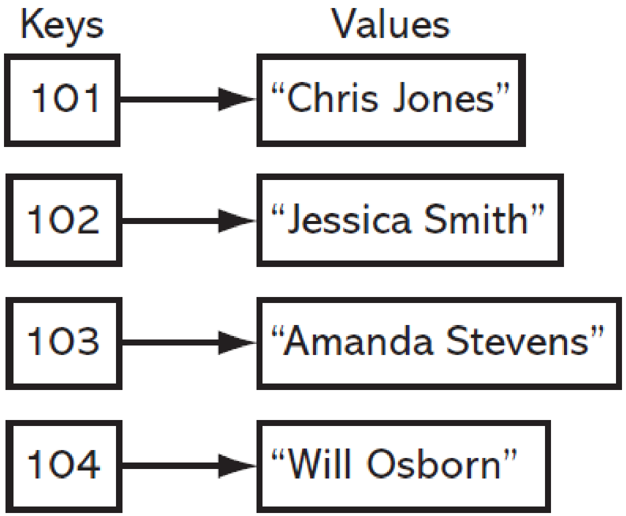
- Each element that is stored in a map has two parts: a *key* and a *value*.

- To retrieve a specific value from a map, you use the key that is associated with that value.

- This is similar to the process of looking up a word in the dictionary, where the words are keys and the definitions are values.

Maps

- Example: a map in which employee IDs are the keys and employee names are the values.
- You use an employee's ID to look up that employee's name.



Keys	Values
101	"Chris Jones"
102	"Jessica Smith"
103	"Amanda Stevens"
104	"Will Osborn"

The `map` Class

- You can use the STL **`map`** class to store *key-value* pairs.

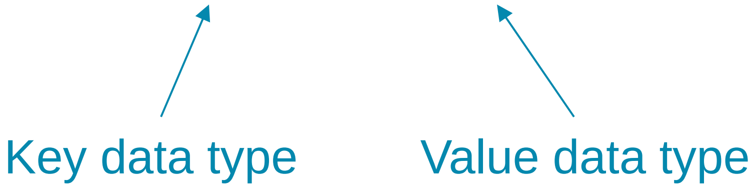
- The keys that are stored in a **`map`** container are unique – no duplicates.

- The **`map`** class is declared in the `<map>` header file.

The `map` Class -- Example

- Example: defining a `map` container to hold employee ID numbers (as `int`s) and their corresponding employee names (as `string`s):

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



Key data type

Value data type

Initializing a `map`

```
map<int, string> employees =
{
   {101, "Chris Jones"},
   {102, "Jessica Smith"},
   {103, "Amanda Stevens"},
   {104, "Will Osborn"}
};
```

- In the first element, the key is 101 and the value is "Chris Jones".
- In the second element, the key is 102 and the value is "Jessica Smith".
- In the third element, the key is 103 and the value is "Amanda Stevens".
- In the fourth element, the key is 104 and the value is "Will Osborn".

The Overloaded [ ] Operator

- You can use the `[]` operator to add new elements to a `map`.
- General format:

```
mapName[key] = value;
```

- This adds the key-value pair to the `map`.
- If the key already exists in the `map`, it's associated value will be changed to *`value`*.

Deleting Elements

- You can use the `erase()` member function to remove a `map` element by its key:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
{
   {101, "Chris Jones"}, {102, "Jessica Smith"},
   {103, "Amanda Stevens"}, {104, "Will Osborn"}
};

// Delete the employee with ID 102.
employees.erase(102);
```

Deletes Jessica Smith from the map

The unordered_map Class

- The unordered_map class is similar to the map class, except in two regards:
  - The keys in an unordered_map are not sorted
  - The unordered_map class has better performance

- You should use the unordered_map class instead of the map class if:
  - You will be making a lot of searches on a large number of elements
  - You are not concerned with retrieving them in key order

- The unordered_map class is declared in the <unordered_map> header file

The `multimap` Class

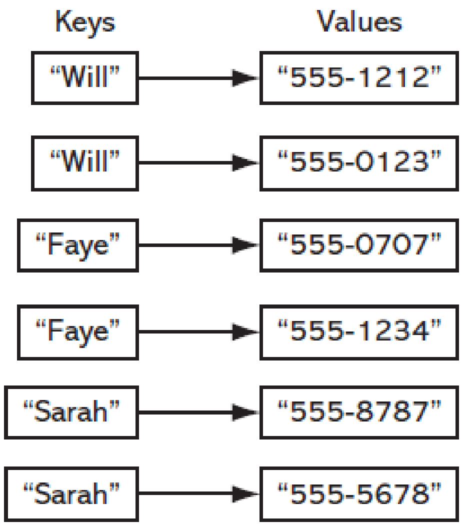
- The `mulitmap` class is a map that allows duplicate keys

- The `mulitmap` class has most of the same member functions as the map class (see Table 17-11 in your textbook)

- The `multimap` class is declared in the `<map>` header file

The `multimap` Class

- Consider a phonebook application where the key is a person's name and the value is that person's phone number.
- A `multimap` container would allow each person to have multiple phone numbers



Keys	Values
"Will"	"555-1212"
"Will"	"555-0123"
"Faye"	"555-0707"
"Faye"	"555-1234"
"Sarah"	"555-8787"
"Sarah"	"555-5678"

Sets

- A *set* is an associative container that is similar to a mathematical set.

- You can use the STL `set` class to create a set container.

- All the elements in a `set` must be unique. No two elements can have the same value.

- The elements in a set are automatically sorted in ascending order.

- The `set` class is declared in the `<set>` header file.

The `set` Class

- You can use the STL `set` class to create a set container.

- The keys that are stored in a `map` container are unique – no duplicates.

The `set` Class

- Example: defining a `set` container to hold integers:

```
set<int> numbers;
```

- Example: defining and initializing a `set` container to hold integers:

```
set<int> numbers = {1, 2, 3, 4, 5};
```

The `set` Class

- A set cannot contain duplicate items.
- If the same value appears more than once in an initialization list, it will be added to the set only one time.
- For example, the following set will contain the values 1, 2, 3, 4, and 5:

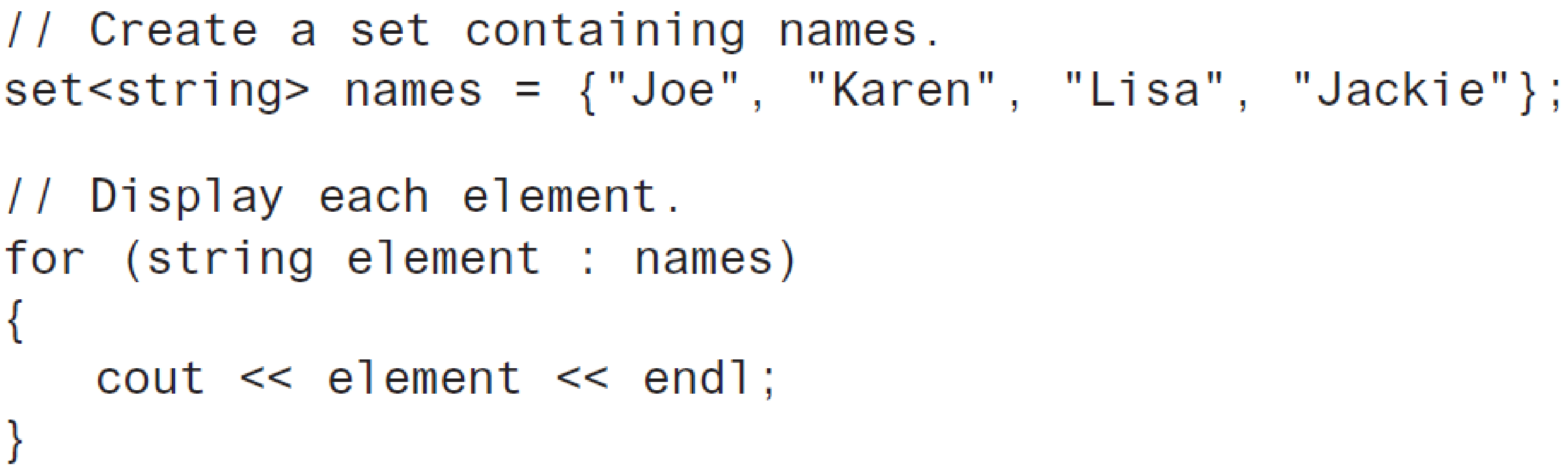
```
set<int> numbers = {1, 1, 2, 2, 3, 4, 5, 5, 5};
```

Adding New Elements to a `set`

- The `insert()` member function adds a new element to a `set`:

```
set<int> numbers;
numbers.insert(10);
numbers.insert(20);
numbers.insert(30);
```

Stepping Through a set With the Range-Based for Loop



```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Display each element.
for (string element : names)
{
   cout << element << endl;
}
```

Using an Iterator With a `set`

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type

- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type

- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type

- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

STL Algorithms

- The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file.

- These functions perform various operations on ranges of elements.

- A range of elements is a sequence of elements denoted by two iterators:
  - The first iterator points to the first element in the range
  - The second iterator points to the end of the range (the element to which the second iterator points is not included in the range).

Categories of Algorithms in the STL

- Min/max algorithms
- Sorting algorithms
- Search algorithms
- Read-only sequence algorithms
- Copying and moving algorithms
- Swapping algorithms
- Replacement algorithms
- Removal algorithms
- Reversal algorithms
- Fill algorithms
- Rotation algorithms
- Shuffling algorithms
- Set algorithms
- Transformation algorithm
- Partition algorithms
- Merge algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm