The Standard Template Library

Kafi Rahman

Assistant Professor @CS
Truman State University

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

17.2

STL Container and Iterator Fundamentals

Containers

- Sequence Containers
  - Stores data sequentially in memory, in a fashion similar to an array
- Associative Containers
  - Stores data in a non-sequential way that makes it faster to locate elements

The array Class Template

- 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

- 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;
```

The array Class Template

- Initializing an array object:

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

array<string, 4> names{"Jamie", "Ashley",
"Doug", "Claire"};
```

The array Class Template

- The array class overloads the [] operator.
- You can use the [] operator to access elements using a subscript, just as you would with a regular array.
- The [] operator does not perform bounds checking. Be careful not to use a subscript that is out of bounds.

The array Class Template

Program 17-1

```
1  #include <iostream>
2  #include <string>
3  #include <array>
4  using namespace std;
5
6  int main()
7  {
8      const int SIZE = 4;
9
10     // Store some names in an array object.
11     array<string, SIZE> names = {"Jamie", "Ashley", "Doug", "Claire"};
12
13     // Display the names.
14     cout << "Here are the names:\n";
15     for (int index = 0; index < names.size(); index++)
16        cout << names[index] << endl;
17
18     return 0;
19 }
```

Program Output

```
Here are the names:
Jamie
Ashley
Doug
Claire
```

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.

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

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)

Iterators

- 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.

Iterators (cont.)

- 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;
```

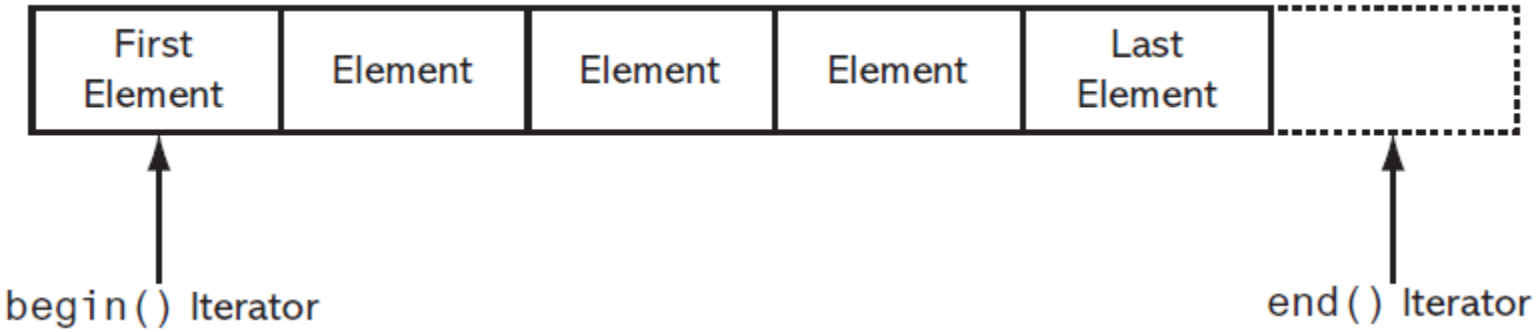
- This defines an iterator named it. The iterator can be used with an array<string, 3> object.

Iterators (cont.)

```
// 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;
```

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

Iterators (cont.)



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

Iterators (cont.)

```
// 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++;
}
```

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

Iterators with auto keyword

- You can use the auto keyword to simplify the definition of an iterator.
- Example:

```
array<string, 3> names = {"Sarah",
"William", "Alfredo"};
auto it = names.begin();
```

Iterators with auto keyword

- The auto keyword specifies that the type of the declared variable will automatically be deduced from its initializer
- With type inference capabilities, we can spend less time having to write out things compiler already knows. For example,

```
auto variable = "string";
cout << variable << endl;
```

Iterators with auto keyword

Program 17-3

```
1  #include <iostream>
2  #include <string>
3  #include <array>
4  using namespace std;
5
6  int main()
7  {
8       const int SIZE = 4;
9
10      // Store some names in an array object.
11      array<string, SIZE> names = {"Jamie", "Ashley", "Doug", "Claire"};
12
13      // Display the names.
14      cout << "Here are the names:\n";
15      for (auto it = names.begin(); it != names.end(); it++)
16         cout << *it << endl;
17
18      return 0;
19 }
```

Iterators with auto keyword

```
array<string, 3> names = {"Olivia", "Emma",
"Drake"};

for (auto it = names.begin(); it <
names.end(); it++)
{       cout << *it << " ";
}


for (auto it = names.end() - 1; it >=
names.begin(); it--)
{       cout << *it << " ";
}
```

Mutable Iterators

```
// Define an array object.
array<int, 5> numbers = {1, 2, 3, 4, 5};

// Define an iterator for the array object.
array<int, 5>::iterator it;

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

// Use the iterator to change the element.
*it = 99;
```

- An iterator gives you read/write access to the element to which the iterator points.
- This is commonly known as a mutable iterator.