Lecture 31:
Standard Template Library (STL)

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 19th, 2010

22.2.1 vector Sequence Container (Cont.)

- An important part of every container is the type of iterator it supports.
- This determines which algorithms can be applied to the container.
- A `vector` supports random-access iterators—i.e., all iterator operations shown in Fig. 22.10 can be applied to a `vector` iterator.
- All STL algorithms can operate on a `vector`.
- The iterators for a `vector` are sometimes implemented as pointers to elements of the `vector`.

22.2.1 vector Sequence Container

- Each STL algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality.
- If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that provides forward iterators, bidirectional iterators or random-access iterators.
- As long as the container supports the algorithm's minimum iterator functionality, the algorithm can operate on the container.

22.2.1 vector Sequence Container

- Figure 22.14 illustrates several functions of the `vector` class template.
- Many of these functions are available in every first-class container.
- You must include header file `<vector>` to use class template `vector`.

```cpp
// Fig. 22.14: Fig22_14.cpp
// Demonstrating Standard Library vector class template.
#include <iostream>
#include <vector> // vector class-template definition
using namespace std;

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

int main()
{
   const int SIZE = 6; // define array size
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
   vector< int > integers; // create vector of ints

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

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

```

Fig. 22.14 | Standard Library `vector` class template. (Part 1 of 3.)

```
    cout << "\nThe size of integers is: " << integers.size()
       << "\nThe capacity of integers is: " << integers.capacity();
    cout << "\n\nOutput array using pointer notation: ";

    // display 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: ";

    // two const reverse iterators
    vector< int >::const_reverse_iterator reverseIterator;
    vector< int >::const_reverse_iterator tempIterator = integers.rend();

    // display vector in reverse order using reverse_iterator
    for ( reverseIterator = integers.rbegin();
       reverseIterator!= tempIterator; ++reverseIterator )
       cout << *reverseIterator << ' ';

    cout << endl;
 } // end main

```

Fig. 22.14 | Standard Library `vector` class template. (Part 2 of 3.)

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

   // display vector elements using const_iterator
   for ( constIterator = integers2.begin();
      constIterator != integers2.end(); ++constIterator )
      cout << *constIterator << ' ';
} // end function printVector
```

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

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Fig. 22.14 | Standard Library `vector` class template. (Part 3 of 3.)

22.2.1 vector Sequence Container

- Line 14 defines an instance called `integers` of class template `vector` that stores `int` values.
- When this object is instantiated, an empty `vector` is created with size 0 (i.e., the number of elements stored in the `vector`) and capacity 0 (i.e., the number of elements that can be stored without allocating more memory to the `vector`).
- Lines 16 and 17 demonstrate the `size` and `capacity` functions; each initially returns 0 for `vector v` in this example.

22.2.1 vector Sequence Container

- Function `size`—available in every container—returns the number of elements currently stored in the container.
- Function `capacity` returns the number of elements that can be stored in the `vector` before the `vector` needs to dynamically resize itself to accommodate more elements.
- Lines 20–22 use function `push_back`—available in all sequence containers—to add an element to the end of the `vector`.
- If an element is added to a full `vector`, the `vector` increases its size—some STL implementations have the `vector` double its capacity.

Performance Tip 22.10

It can be wasteful to double a `vector`*'s size when more space is needed. For example, a full* `vector` *of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added. This leaves 999,999 unused elements. You can use* `resize` *and* `reserve` *to control space usage better.*

22.2.1 vector Sequence Container

- Lines 24 and 25 use `size` and `capacity` to illustrate the new size and capacity of the `vector` after the three `push_back` operations.
- Function `size` returns 3—the number of elements added to the `vector`.
- Function `capacity` returns 4, indicating that we can add one more element before the `vector` needs to add more memory.
- When we added the first element, the `vector` allocated space for one element, and the size became 1 to indicate that the `vector` contained only one element.

22.2.1 vector Sequence Container

- When we added the second element, the capacity doubled to 2 and the size became 2 as well.
- When we added the third element, the capacity doubled again to 4.
- So we can actually add another element before the `vector` needs to allocate more space.
- When the `vector` eventually fills its allocated capacity and the program attempts to add one more element to the `vector`, the `vector` will double its capacity to 8 elements.
- The manner in which a `vector` grows to accommodate more elements—a time consuming operation—is not specified by the C++ Standard Document.

22.2.1 vector Sequence Container

- C++ library implementors use various clever schemes to minimize the overhead of resizing a `vector`.
- Hence, the output of this program may vary, depending on the version of `vector` that comes with your compiler.
- Some library implementors allocate a large initial capacity.
- If a `vector` stores a small number of elements, such capacity may be a waste of space.
- However, it can greatly improve performance if a program adds many elements to a `vector` and does not have to reallocate memory to accommodate those elements.
- This is a classic space–time trade-off.

22.2.1 vector Sequence Container

- Library implementors must balance the amount of memory used against the amount of time required to perform various `vector` operations.
- Lines 29–30 demonstrate how to output the contents of an array using pointers and pointer arithmetic.
- Line 33 calls function `printVector` (defined in lines 49–57) to output the contents of a `vector` using iterators.
- Function template `printVector` receives a `const` reference to a `vector` (`integers2`) as its argument.
- Line 51 defines a `const_iterator` called `constIterator` that iterates through the `vector` and outputs its contents.
- Notice that the declaration in line 51 is prefixed with the keyword `typename`.

22.2.1 vector Sequence Container

- A `const_iterator` enables the program to read the elements of the `vector`, but does not allow the program to modify the elements.
- The `for` statement in lines 54–56 initializes `constIterator` using `vector` member function `begin`, which returns a `const_iterator` to the first element in the `vector`—there is another version of `begin` that returns an `iterator` that can be used for non-`const` containers.
- A `const_iterator` is returned because the identifier `integers2` was declared `const` in the parameter list of function `printVector`.
- The loop continues as long as `constIterator` has not reached the end of the `vector`.

22.2.1 vector Sequence Container

- Line 37 declares a `const_reverse_iterator` that can be used to iterate through a `vector` backward.
- Line 38 declares a `const_reverse_iterator` variable `tempIterator` and initializes it to the iterator returned by function rend (i.e., the iterator for the ending point when iterating through the container in reverse).
- All first-class containers support this type of iterator.
- Lines 41–43 use a `for` statement similar to that in function `printVector` to iterate through the `vector`.
- In this loop, function rbegin (i.e., the iterator for the starting point when iterating through the container in reverse) and `tempIterator` delineate the range of elements to output.
- As with functions `begin` and `end`, `rbegin` and `rend` can return a `const_reverse_iterator` or a `reverse_iterator`, based on whether or not the container is constant.

Questions

?