Lecture 30: Standard Template Library (STL)

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 18th, 2010

digitalblasphemy

22.1 Introduction to the Standard Template Library (STL)

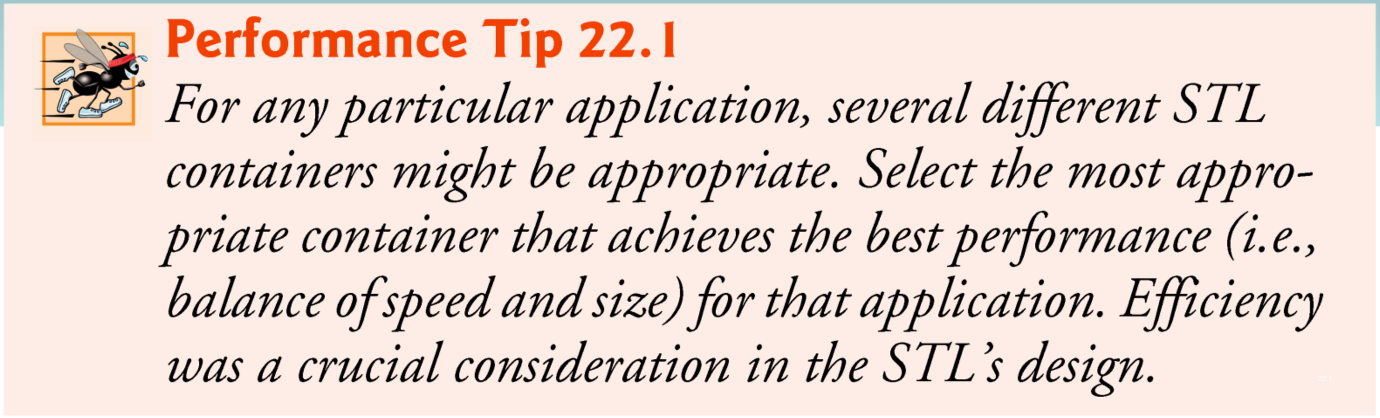
- We’ve repeatedly emphasized the importance of software reuse.
- Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library.
- The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

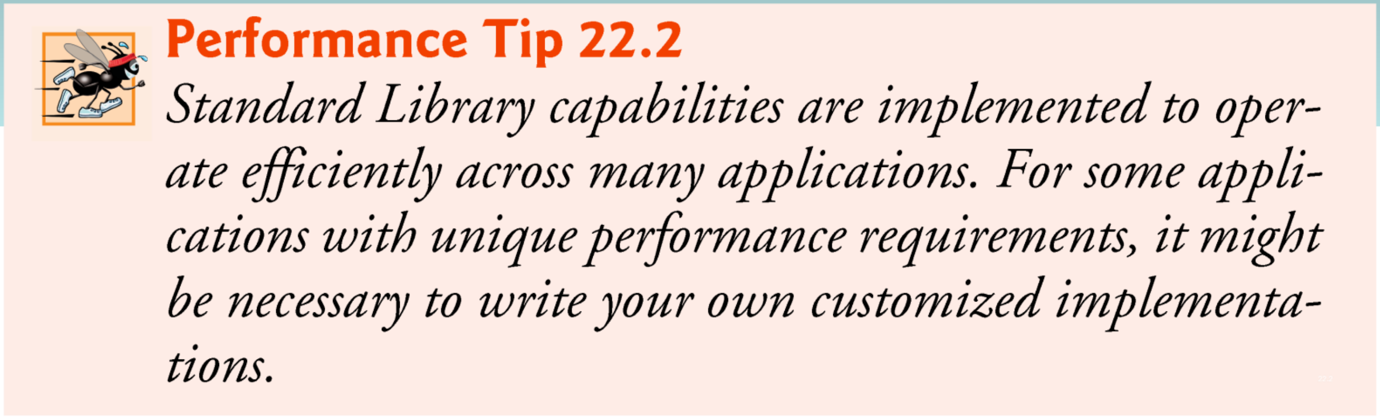
- As you'll see, the STL was conceived and designed for performance and flexibility.
- This chapter introduces the STL and discusses its three key components—containers (popular templatized data structures), iterators and algorithms.
- The STL containers are data structures capable of storing objects of almost any data type (there are some restrictions).
- We'll see that there are three styles of container classes—first-class containers, adapters and near containers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Performance Tip 22.1

For any particular application, several different STL containers might be appropriate. Select the most appropriate container that achieves the best performance (i.e., balance of speed and size) for that application. Efficiency was a crucial consideration in the STL's design.



Performance Tip 22.2

Standard Library capabilities are implemented to operate efficiently across many applications. For some applications with unique performance requirements, it might be necessary to write your own customized implementations.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

- Each STL container has associated member functions.
- A subset of these member functions is defined in all STL containers.
- We illustrate most of this common functionality in our examples of STL containers `vector` (a dynamically resizable array which we introduced in Chapter 7), `list` (a doubly linked list) and deque (a double-ended queue, pronounced “deck”).
- We introduce container-specific functionality in examples for each of the other STL containers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

- STL iterators, which have properties similar to those of pointers, are used by programs to manipulate the STL-container elements.
- In fact, standard arrays can be manipulated by STL algorithms, using standard pointers as iterators.
- We'll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement.
- There are five categories of iterators, each of which we discuss in Section 22.1.2 and use throughout this chapter.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

- STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements (or entire containers).
- The STL provides approximately 70 algorithms.
- Most of them use iterators to access container elements.
- Each algorithm has minimum requirements for the types of iterators that can be used with it.
- We’ll see that each first-class container supports specific iterator types, some more powerful than others.
- A container’s supported iterator type determines whether the container can be used with a specific algorithm.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

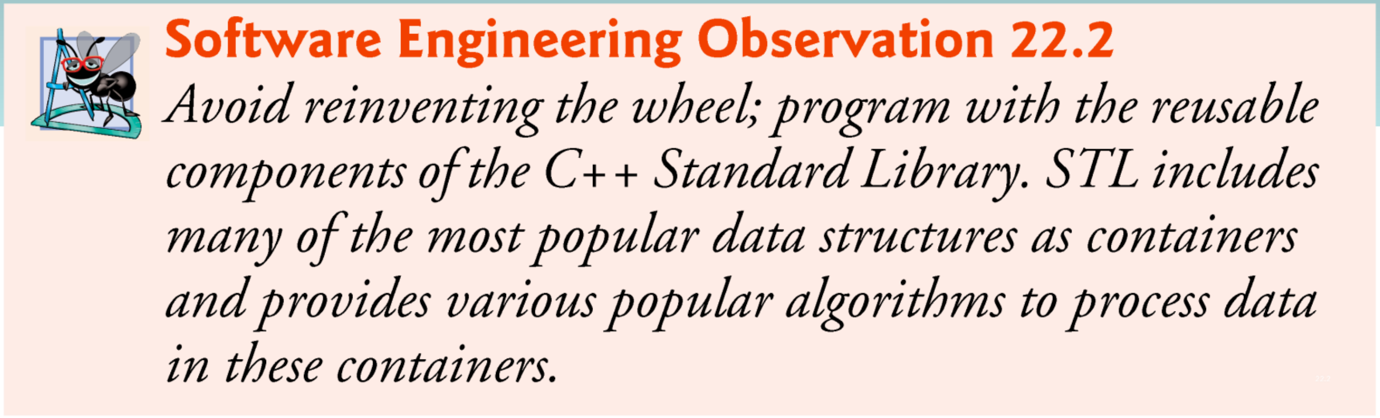
- Iterators encapsulate the mechanism used to access container elements.
- This encapsulation enables many of the STL algorithms to be applied to several containers without regard for the underlying container implementation.
- As long as a container's iterators support the minimum requirements of the algorithm, then the algorithm can process that container's elements.
- This also enables you to create new algorithms that can process the elements of multiple container types.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1 Introduction to the Standard Template Library (STL) (Cont.)

- In Chapter 20, we studied data structures.
- We built linked lists, queues, stacks and trees.
- We carefully wove link objects together with pointers.
- Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints.
- Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work.
- An advantage of the STL is that you can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Software Engineering Observation 22.2

Avoid reinventing the wheel; program with the reusable components of the C++ Standard Library. STL includes many of the most popular data structures as containers and provides various popular algorithms to process data in these containers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 22.1

When programming pointer-based data structures and algorithms, we must do our own debugging and testing to be sure our data structures, classes and algorithms function properly. It's easy to make errors when manipulating pointers at this low level. Memory leaks and memory-access violations are common in such custom code. The prepackaged, templatized containers of the STL are sufficient for most programmers. Using the STL helps you reduce testing and debugging time. One caution is that, for large projects, template compile time can be significant.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.1 Introduction to Containers

- The STL container types are shown in Fig. 22.1.
- The containers are divided into three major categories—sequence containers, associative containers and container adapters.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).

Fig. 22.1 | Standard Library container classes. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Standard Library container class	Description
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 22.1 | Standard Library container classes. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.1 Introduction to Containers (Cont.)

- The sequence containers represent linear data structures, such as vectors and linked lists.
- Associative containers are nonlinear containers that typically can locate elements stored in the containers quickly.
  - Such containers can store sets of values or key/value pairs.
- The sequence containers and associative containers are collectively referred to as the first-class containers.
- As we saw in Chapter 20, stacks and queues actually are constrained versions of sequential containers.
- For this reason, STL implements stacks and queues as container adapters that enable a program to view a sequential container in a constrained manner.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.1 Introduction to Containers (Cont.)

- Most STL containers provide similar functionality.
- Many generic operations, such as member function `size`, apply to all containers, and other operations apply to subsets of similar containers.
- This encourages extensibility of the STL with new classes.
- Figure 22.2 describes the functions common to all Standard Library containers.
- [*Note: Overloaded operators* *`operator<`*, *`operator<=`*, *`operator>`*, *`operator>=`*, *`operator==`* *and* *`operator!=`* *are not provided for* *`priority_queue`s.]*

Member function	Description
default constructor	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are no elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.

Fig. 22.2 | Common member functions for most STL containers. (Part 1 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Member function	Description
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.1 Introduction to Containers (Cont.)

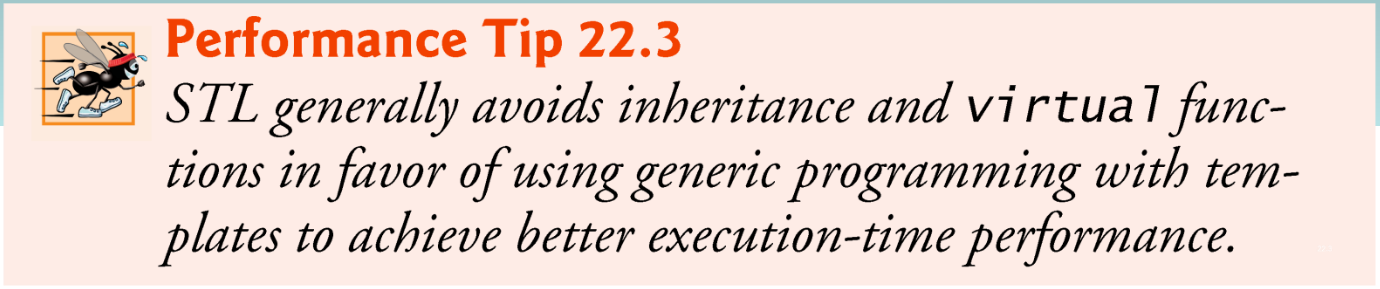
- The header files for each of the Standard Library containers are shown in Fig. 22.3.
- The contents of these header files are all in `namespace std`.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Standard Library container header files	
`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

Fig. 22.3 | Standard Library container header files.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Performance Tip 22.3

STL generally avoids inheritance and `virtual` *functions in favor of using generic programming with templates to achieve better execution-time performance.*

Portability Tip 22.1

Programming with STL will enhance the portability of your code.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.1 Introduction to Containers (Cont.)

- When preparing to use an STL container, it's important to ensure that the type of element being stored in the container supports a minimum set of functionality.
- When an element is inserted into a container, a copy of that element is made.
- For this reason, the element type should provide its own copy constructor and assignment operator.
- [*Note: This is required only if default memberwise copy and default memberwise assignment do not perform proper copy and assignment operations for the element type.]*
- Also, the associative containers and many algorithms require elements to be compared.
- For this reason, the element type should provide an equality operator (==) and a less-than operator (<).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 22.3

The STL containers do not require their elements to be comparable with the equality and less-than operators unless a program uses a container member function that must compare the container elements (e.g., the `sort` *function in class* `list`*). Some pre-standard C++ compilers are not capable of ignoring parts of a template that are not used in a particular program. On compilers with this problem, you may not be able to use the STL containers with objects of classes that do not define overloaded less-than and equality operators.*

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators

- Iterators have many features in common with pointers and are used to point to the elements of first-class containers (and for a few other purposes, as we'll see).
- Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container.
- Certain iterator operations are uniform across containers.
- For example, the dereferencing operator (*) dereferences an iterator so that you can use the element to which it points.
- The ++ operation on an iterator moves it to the next element of the container (much as incrementing a pointer into an array aims the pointer at the next element of the array).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- STL first-class containers provide member functions `begin` and `end`.
- Function begin returns an iterator pointing to the first element of the container.
- Function end returns an iterator pointing to the first element past the end of the container (an element that doesn’t exist).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- If iterator `i` points to a particular element, then `++i` points to the "next" element and `*i` refers to the element pointed to by `i`.
- The iterator resulting from `end` is typically used in an equality or inequality comparison to determine whether the "moving iterator" (`i` in this case) has reached the end of the container.
- An object of type `iterator` refers to a container element that can be modified.
- An object of type `const_iterator` refers to a container element that cannot be modified.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- `We use iterators with sequences ranges).`
- These sequences can be in containers, or they can be input sequences or output sequences.
- The program of Fig. 22.5 demonstrates input from the standard input (a sequence of data for input into a program), using an `istream_iterator`, and output to the standard output (a sequence of data for output from a program), using an `ostream_iterator`.
- The program inputs two integers from the user at the keyboard and displays the sum of the integers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

```cpp
// Fig. 22.5: Fig22_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>
#include <iterator> // ostream_iterator and istream_iterator
using namespace std;

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

   // create istream_iterator for reading int values from cin
   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

   // create ostream_iterator for writing int values to cout
   ostream_iterator< int > outputInt( cout );

```

Fig. 22.5 | Input and output stream iterators. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

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

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

Fig. 22.5 | Input and output stream iterators. (Part 2 of 2.)

22.1.2 Introduction to Iterators (Cont.)

- Line 12 creates an `istream_iterator` that is capable of extracting (inputting) `int` values in a type-safe manner from the standard input object `cin`.
- Line 14 dereferences iterator `inputInt` to read the first integer from `cin` and assigns that integer to `number1`.
- The dereferencing operator `*` applied to `inputInt` gets the value from the stream associated with `inputInt`; this is similar to dereferencing a pointer.
- Line 15 positions iterator `inputInt` to the next value in the input stream.
- Line 16 inputs the next integer from `inputInt` and assigns it to `number2`.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- Line 19 creates an `ostream_iterator` that is capable of inserting (outputting) `int` values in the standard output object `cout`.
- Line 22 outputs an integer to `cout` by assigning to `*outputInt` the sum of `number1` and `number2`.
- Notice the use of the dereferencing operator `*` to use `*outputInt` as an *lvalue in the assignment statement*.
- If you want to output another value using `outputInt`, the iterator must be incremented with `++` (both the prefix and postfix increment can be used, but the prefix form should be preferred for performance reasons).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

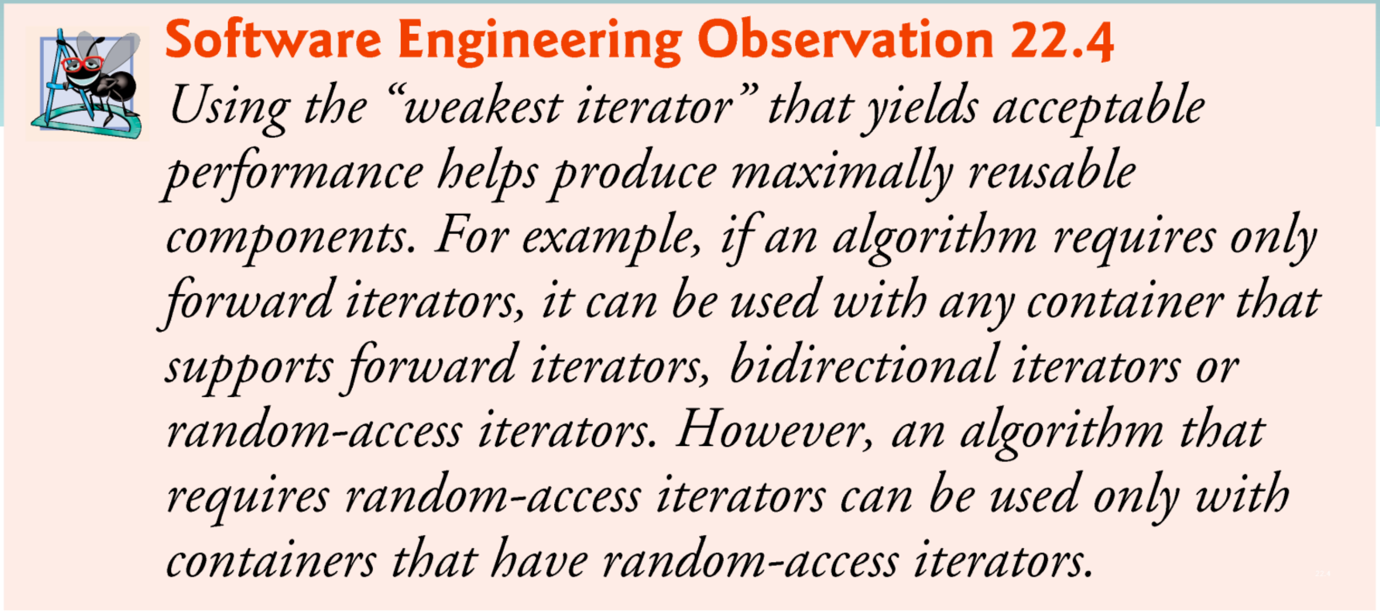
Fig. 22.6 | Iterator categories.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- The iterator category that each container supports determines whether that container can be used with specific algorithms in the STL.
- Containers that support random-access iterators can be used with all algorithms in the STL.
- As we'll see, pointers into arrays can be used in place of iterators in most STL algorithms, including those that require random-access iterators.
- Figure 22.8 shows the iterator category of each of the STL containers.
- The first-class containers (`vector`s, `deque`s, `list`s, `set`s, `multiset`s, `map`s and `multimap`s), `string`s and arrays are all traversable with iterators.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Software Engineering Observation 22.4

Using the "weakest iterator" that yields acceptable performance helps produce maximally reusable components. For example, if an algorithm requires only forward iterators, it can be used with any container that supports forward iterators, bidirectional iterators or random-access iterators. However, an algorithm that requires random-access iterators can be used only with containers that have random-access iterators.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

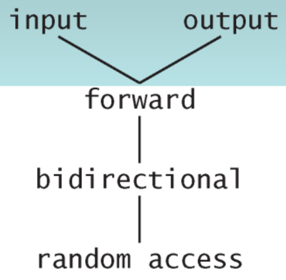


Fig. 22.7 | Iterator category hierarchy.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Fig. 22.8 | Iterator types supported by each container.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.2 Introduction to Iterators (Cont.)

- Figure 22.9 shows the predefined iterator `typedef`s that are found in the class definitions of the STL containers.
- Not every `typedef` is defined for every container.
- We use `const` versions of the iterators for traversing read-only containers.
- We use reverse iterators to traverse containers in the reverse direction.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Predefined typedefs for iterator types	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

Fig. 22.9 | Iterator typedefs.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 22.3

Operations performed on a `const_iterator` *return* `const` *references to prevent modification to elements of the container being manipulated. Using* `const_iterators` *where appropriate is another example of the principle of least privilege.*

22.1.2 Introduction to Iterators (Cont.)

- Figure 22.10 shows some operations that can be performed on each iterator type.
- The operations for each iterator type include all operations preceding that type in the figure.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 1 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
p += i	Increment the iterator p by i positions.
p -= i	Decrement the iterator p by i positions.
p + i *or* i + p	Expression value is an iterator positioned at p incremented by i positions.
p - i	Expression value is an iterator positioned at p decremented by i positions.
p - p1	Expression value is an integer representing the distance between two elements in the same container.
p[i]	Return a reference to the element offset from p by i positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 2 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Iterator operation	Description
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 3 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.3 Introduction to Algorithms

- STL algorithms can be used generically across a variety of containers.
- STL provides many algorithms you'll use frequently to manipulate containers.
- Inserting, deleting, searching, sorting and others are appropriate for some or all of the STL containers.
- The STL includes approximately 70 standard algorithms.
- The algorithms operate on container elements only indirectly through iterators.
- Many algorithms operate on sequences of elements defined by pairs of iterators—one pointing to the first element of the sequence and one pointing to one element past the last element.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.3 Introduction to Algorithms (Cont.)

- Algorithms often return iterators that indicate the results of the algorithms.
- Algorithm `find`, for example, locates an element and returns an iterator to that element.
- If the element is not found, `find` returns the "one past the `end`" iterator that was passed in to define the end of the range to be searched, which can be tested to determine whether an element was not found.
- The `find` algorithm can be used with any first-class STL container.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.3 Introduction to Algorithms (Cont.)

- STL algorithms create yet another opportunity for reuse—using the rich collection of popular algorithms can save you much time and effort.
- If an algorithm uses less powerful iterators, the algorithm can also be used with containers that support more powerful iterators.
- Some algorithms demand powerful iterators; e.g., `sort` demands random-access iterators.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 22.5

The STL is extensible. It's straightforward to add new algorithms and to do so without changes to STL containers.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 22.6

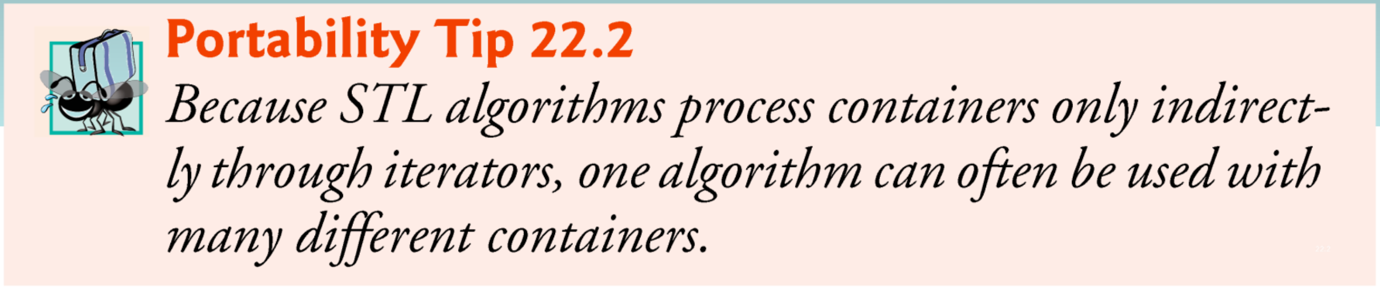
The STL is implemented concisely. The algorithms are separated from the containers and operate on elements of the containers only indirectly through iterators. This separation makes it easier to write generic algorithms applicable to many container classes.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 22.7

STL algorithms can operate on STL containers and on pointer-based, C-like arrays.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Portability Tip 22.2

Because STL algorithms process containers only indirectly through iterators, one algorithm can often be used with many different containers.

22.1.3 Introduction to Algorithms (Cont.)

- Figure 22.11 shows many of the mutating-sequence algorithms—i.e., the algorithms that result in modifications of the containers to which the algorithms are applied.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Mutating-sequence algorithms			
`copy`	`partition`	`replace_copy`	`stable_partition`
`copy_backward`	`random_shuffle`	`replace_copy_if`	`swap`
`fill`	`remove`	`replace_if`	`swap_ranges`
`fill_n`	`remove_copy`	`reverse`	`transform`
`generate`	`remove_copy_if`	`reverse_copy`	`unique`
`generate_n`	`remove_if`	`rotate`	`unique_copy`
`iter_swap`	`replace`	`rotate_copy`	

Fig. 22.11 | Mutating-sequence algorithms.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.1.3 Introduction to Algorithms (Cont.)

- Figure 22.12 shows many of the nonmodifying sequence algorithms—i.e., the algorithms that do not result in modifications of the containers to which they're applied.
- Figure 22.13 shows the numerical algorithms of the header file `<numeric>`.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Nonmodifying sequence algorithms			
adjacent_find	equal	find_end	mismatch
count	find	find_first_of	search
count_if	find_each	find_if	search_n

Fig. 22.12 | Nonmodifying sequence algorithms.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Numerical algorithms from header file `<numeric>`	
`accumulate`	`partial_sum`
`inner_product`	`adjacent_difference`

Fig. 22.13 | Numerical algorithms from header file `<numeric>`.

22.2 Sequence Containers

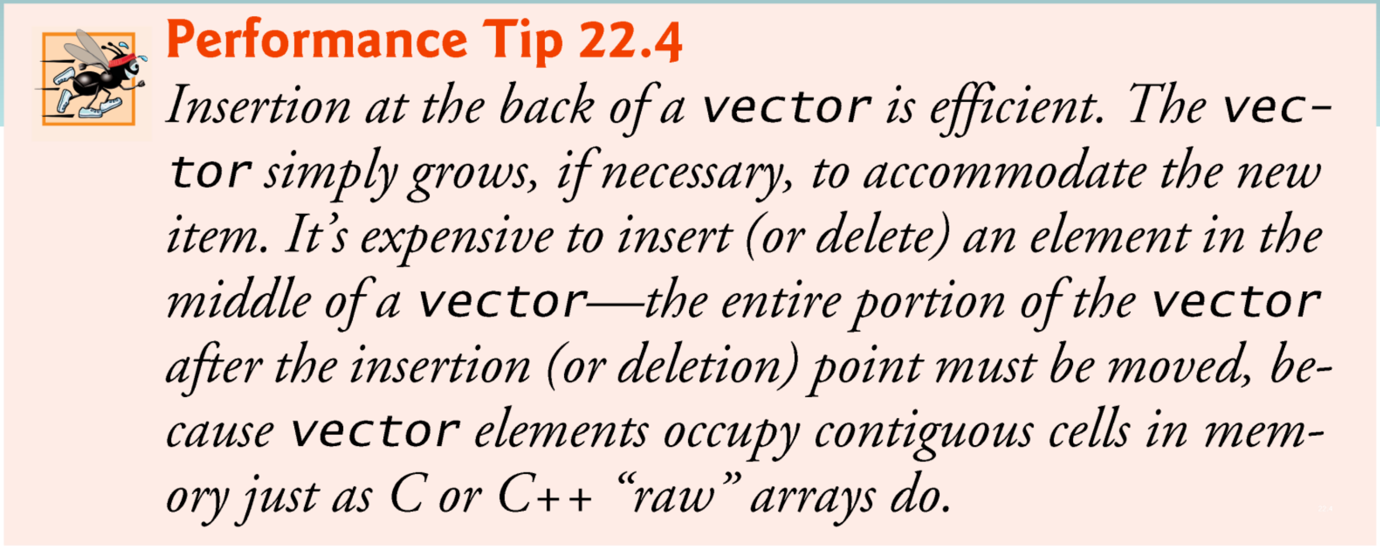
- The C++ Standard Template Library provides three sequence containers—`vector`, `list` and `deque`.
- Class template `vector` and class template `deque` both are based on arrays.
- Class template `list` implements a linked-list data structure similar to our `List` class presented in Chapter 20, but more robust.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.2 Sequence Containers (Cont.)

- One of the most popular containers in the STL is `vector`.
- Recall that we introduced class template `vector` in Chapter 7 as a more robust type of array.
- A `vector` changes size dynamically.
- Unlike C and C++ “raw” arrays (see Chapter 7), `vector`s can be assigned to one another.
- This is not possible with pointer-based, C-like arrays, because those array names are constant pointers and cannot be the targets of assignments.
- Just as with C arrays, `vector` subscripting does not perform automatic range checking, but class template `vector` does provide this capability via member function `at` (also discussed in Chapter 7).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Performance Tip 22.4

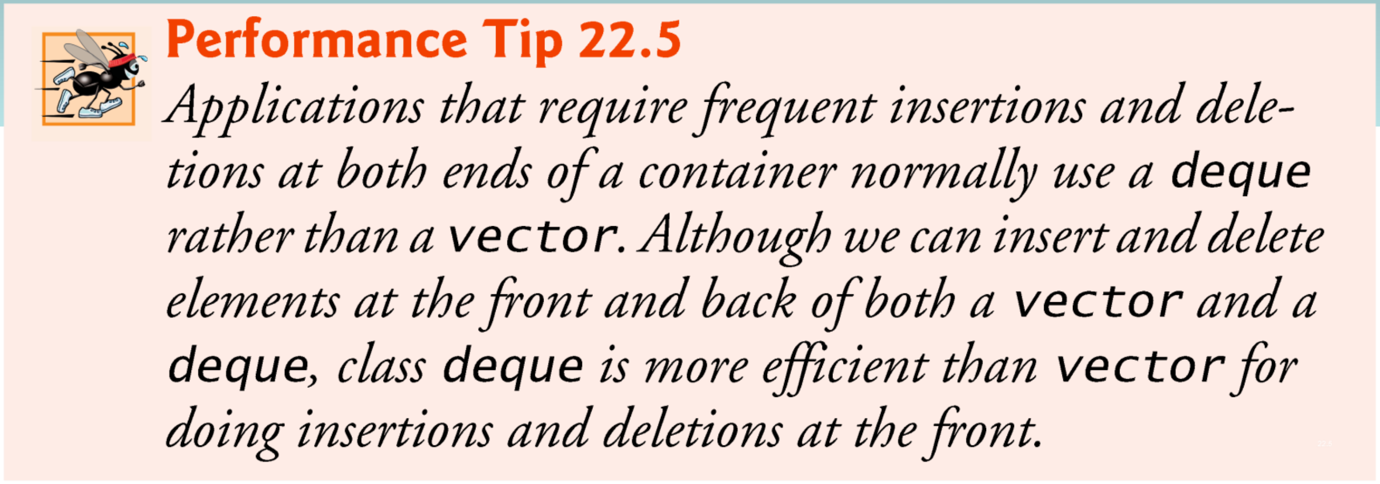
Insertion at the back of a `vector` *is efficient. The* `vector` *simply grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a* `vector`*—the entire portion of the* `vector` *after the insertion (or deletion) point must be moved, because* `vector` *elements occupy contiguous cells in memory just as C or C++ "raw" arrays do.*

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.2 Sequence Containers (Cont.)

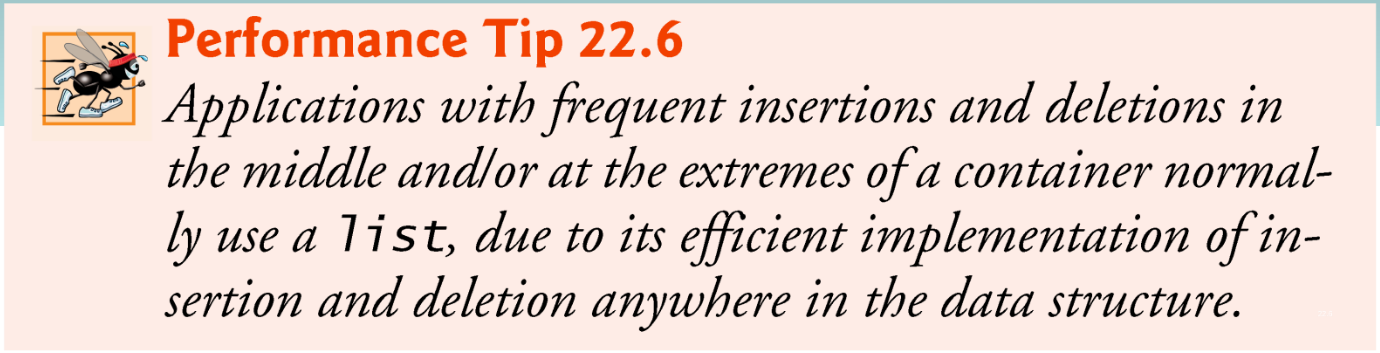
- Figure 22.2 presented the operations common to all the STL containers.
- Beyond these operations, each container typically provides a variety of other capabilities.
- Many of these capabilities are common to several containers, but they're not always equally efficient for each container.
- You must choose the container most appropriate for the application.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.



Performance Tip 22.5

Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` *rather than a* `vector`*. Although we can insert and delete elements at the front and back of both a* `vector` *and a* `deque`*, class* `deque` *is more efficient than* `vector` *for doing insertions and deletions at the front.*



Performance Tip 22.6

Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a `list`*, due to its efficient implementation of insertion and deletion anywhere in the data structure.*

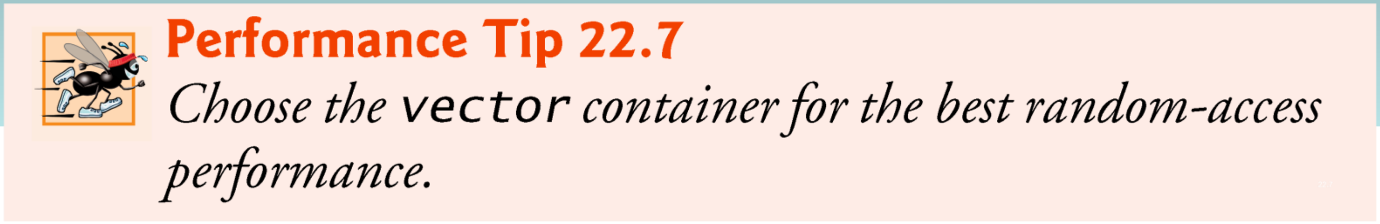
22.2 Sequence Containers (Cont.)

- In addition to the common operations described in Fig. 22.2, the sequence containers have several other common operations—`front` to return a reference to the first element in a non-empty container, `back` to return a reference to the last element in a non-empty container, `push_back` to insert a new element at the end of the container and `pop_back` to remove the last element of the container.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

22.2.1 vector Sequence Container

- Class template `vector` provides a data structure with contiguous memory locations.
- This enables efficient, direct access to any element of a vector via the subscript operator `[]`, exactly as with a C or C++ "raw" array.
- Class template `vector` is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted.
- When a `vector`'s memory is exhausted, the `vector` allocates a larger contiguous area of memory, copies the original elements into the new memory and deallocates the old memory.



Performance Tip 22.7

Choose the `vector` *container for the best random-access performance.*

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Performance Tip 22.8

Objects of class template `vector` *provide rapid indexed access with the overloaded subscript operator* `[]` *because they're stored in contiguous memory like a C or C++ raw array.*

Performance Tip 22.9

It's faster to insert many elements at once than one at a time.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Questions

?

©1992-2010 by Pearson Education, Inc. All Rights Reserved.