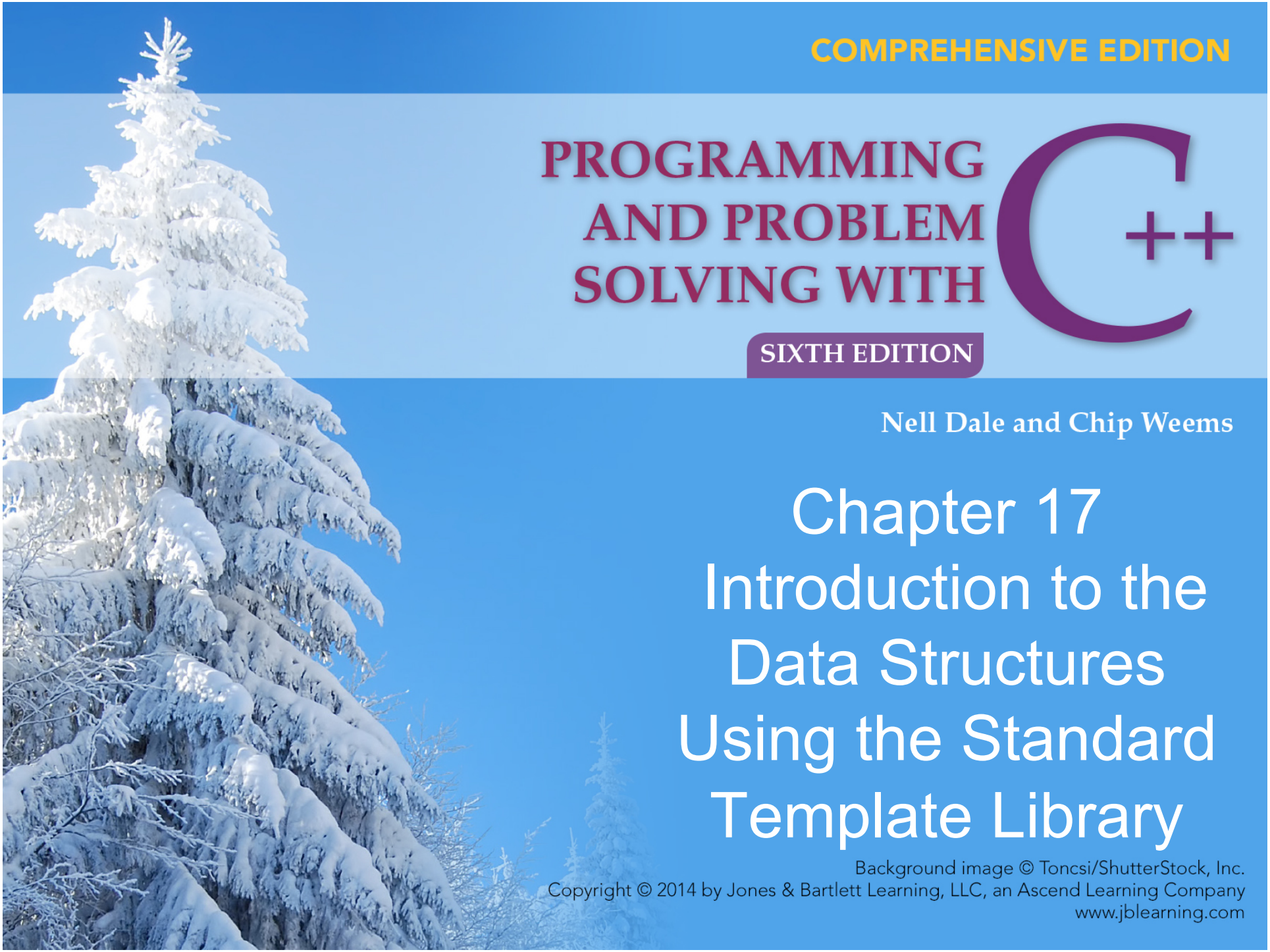


COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 17 Introduction to the Data Structures Using the Standard Template Library

Chapter Topics

- The distinction between Standard Template Library (STL) and data structures in the abstract sense
- Containers and Algorithms of the Standard Template Library
- Working with STL iterators
- Operation of stacks, queues, and deques

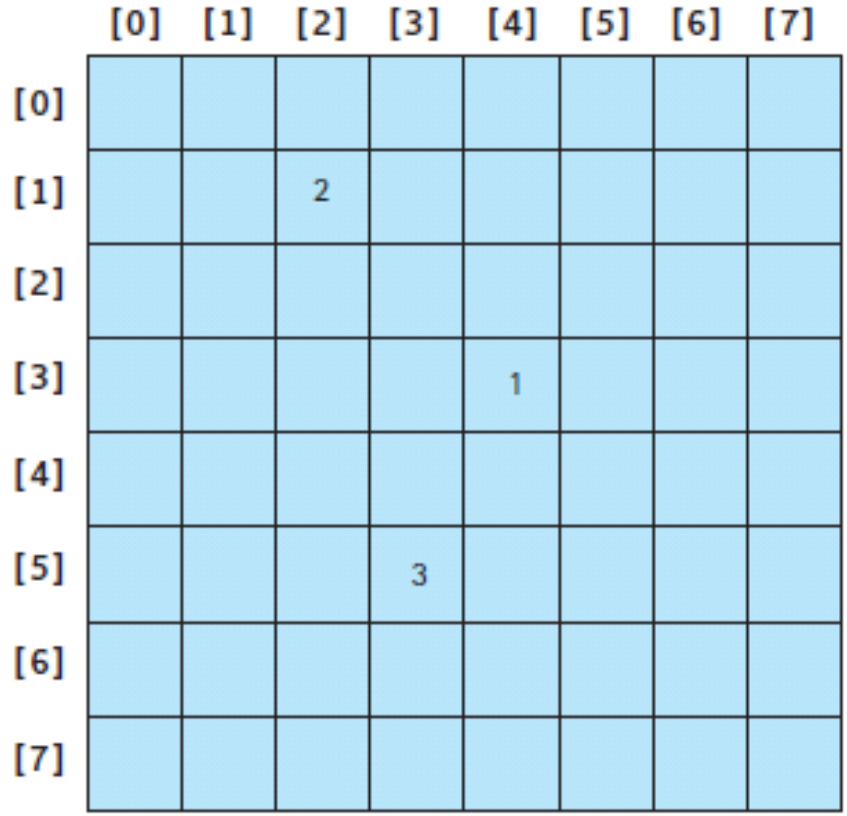
Chapter Topics

- Using the basic STL sequence containers: **vector, list, and deque**
- The structure and operations associated with tree and hash table abstract types
- Using the STL associative containers **set** and **map** as abstract data types

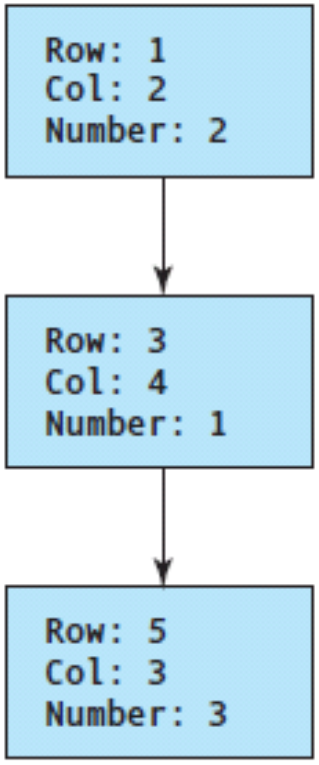
Abstract Data Structures vs. Implementations

- Every data structure has two aspects:
  - its abstract (conceptual) form and
  - its implementation
- Advantage of being able to use data and control abstraction to divorce these two aspects in object-oriented programming:
- Client code can use an comprehensible natural representation that saves computer memory space through an efficient underlying implementation

A Linked Implementation of a Sparsely Filled Array Conserves Space



Array Implementation



Linked Implementation

Additional Linear Structures

- Differences in linear data structures are based on how they are accessed
- Arrays and lists support very general access
- If access is restricted in some way, we can define a data structure with a simpler interface, e.g., **stacks and cues**
- Also, in some cases the restrictions permit the programmer to create a more efficient implementation

Containers of the Standard Template Library (STL)

- STL composed primarily of templates
- Two main sections of the library: **containers** and **algorithms**
- C++ 's templates for 8 generic structured data types of the containers portion are summarized on p. 879 of Ch. 14
- We will describe these 8 template classes:

  stack, list, queue, priority_queue, vector, set, and map

Declaring and Specializing STL Template Classes

To declare and specialize any of these above template classes, use its name, followed by the type in angle brackets

Example:

```
list<string> strList; // Create an empty list of strings
vector<int> intVec; // Create an empty vector of ints
```

Containers Portion of STL

- Type of container's elements can either be built-in type or a class
- STL requires that element objects support at least a minimal subset of the overloaded relational operators
- For most compilers, overloading < and == is sufficient, but a few require more of the operators to be overloaded

Algorithms Portion of STL

- Algorithms portion of the STL has a large number of function templates used in many different contexts
- Table on pp. 880 of Ch. 17 summarizes the more commonly used functions: `copy`, `copy_backward`, `equal`, `find`, `max`, `min`, `random_shuffle`, `replace`, `search`, `set_difference`, and `set_intersection`, `set_union`, `sort`, and `transform`
- Not all of these above functions can be applied to every container

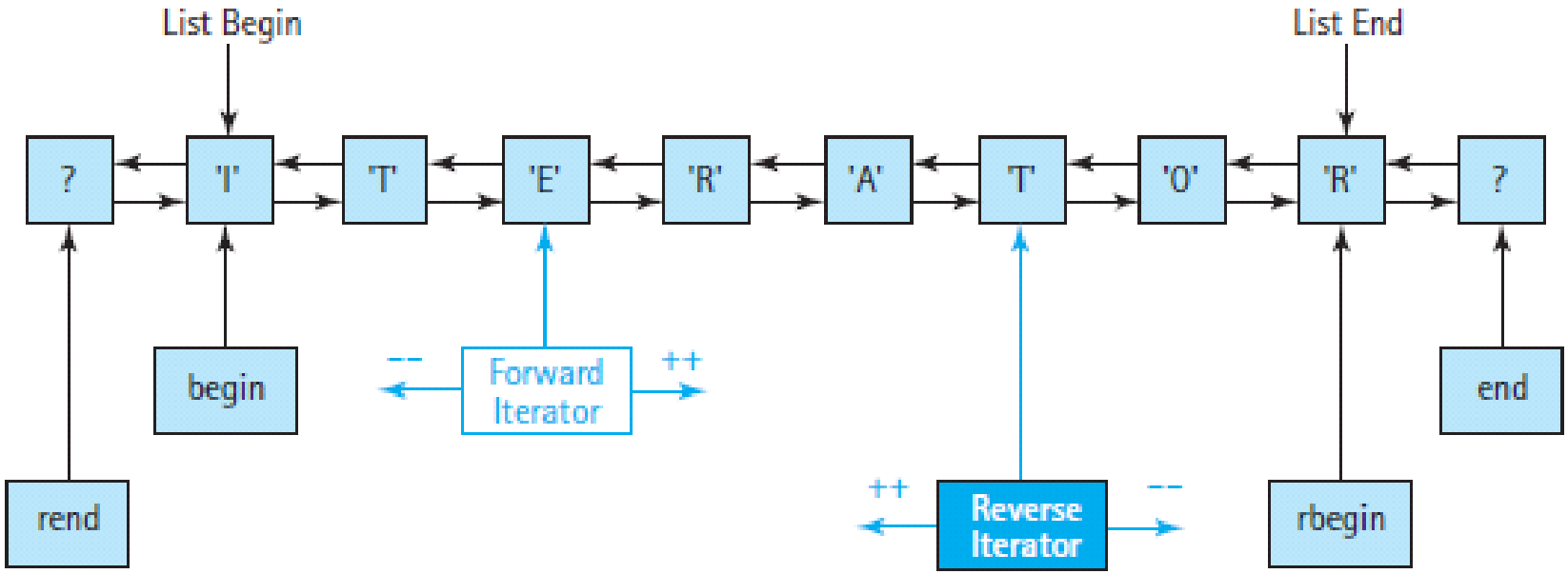
Iterators in the STL

- In the STL, the **iterator** is an object that points to some place in a container
- An iterator has extra functionality including operations that cause it to move in a step-by-step fashion through an STL container
- Two categories of iterators in the STL: forward and reverse

Iterators in the STL

- Use the `++` operator to advance a forward iterator and `--` to shift the iterator backward
- Iterators returned by **`rbegin`** and **`rend`** point to the end and the beginning of a container, respectively
- Using `++` with a reverse iterator moves it from the back to the front and applying `--` causes a move to the back

Forward and Reverse Iterators Associated with a `List`



Iterators in the STL

- Design of these iterators meant to facilitate traversing a container with a For loop
- With random-access containers (such as `deque` and `vector`), + and - can be applied to iterators compute the location of another element
- Iterators remain valid only as long as the contents of the container do not change
- When an element is inserted into or deleted from a container, its iterators become invalid

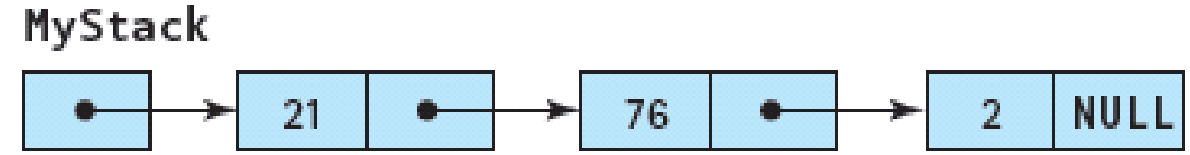
Stacks

- A **stack** is a data structure that can be accessed from only one end
- With a stack, one has the ability to insert an element at the top (as the first element) and remove the top (first) element
- Accountants call this property, LIFO, "last in, first out"
- Real-life examples include plate holder stacked in a cafeteria or cars in a noncircular driveway

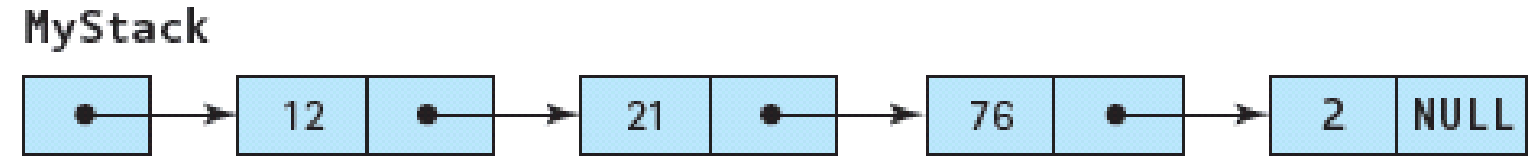
Stacks

- Term `push` is used for insertion operation and the term `pop` is used for the deletion operation
- In some implementations of a stack, `pop` also returns the top element
- In other implementations, a separate operation, `top`, retrieves the top value and `pop` simply deletes it

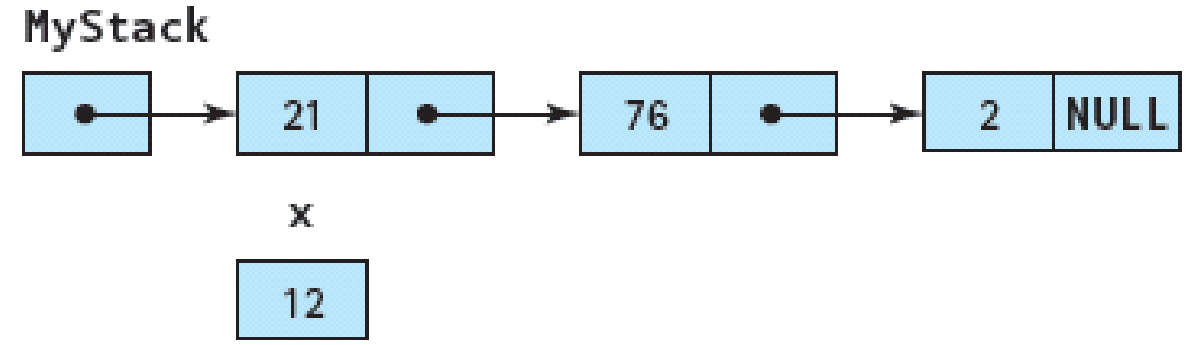
A Stack Showing Push, Top, and Pop Operations



a. Initial value of stack



b. `mystack.push(12)` pushes a new element on the stack with a value of 12



c. `x = mystack.top();` returns an element from the stack and
`mystack.pop();` pops it from the top

Uses of Stacks in Systems Software

- C++ uses stacks to keep track of nested method calls
- Compiler uses a stack to translate arithmetic expressions
- Stacks used whenever programmer wants to remember a sequence of objects or actions in the reverse order from their original occurrence

Example of Use of a Stack

An algorithm to read in a file containing a series of lines and printing out the lines in reverse order using a stack of strings shown below:

```
Create stack
Create file in
while in

    stack.push(in.getline)
while not stack. empty
    print stack.top
    stack.pop
```

Algorithm Example

- Algorithm on previous slide illustrates, one needs a way to test whether the stack is empty
- Also, one may desire to find the size of a stack
- Even with these additions, the stack interface has only five responsibilities making it a simple abstract type

Queues

- A **queue**–a data structure in which elements are entered at one end and removed from the other end
- Accountants call this property, FIFO, "first in, first out"
- Examples include a waiting line in a bank or supermarket and a line of cars on a one-way street

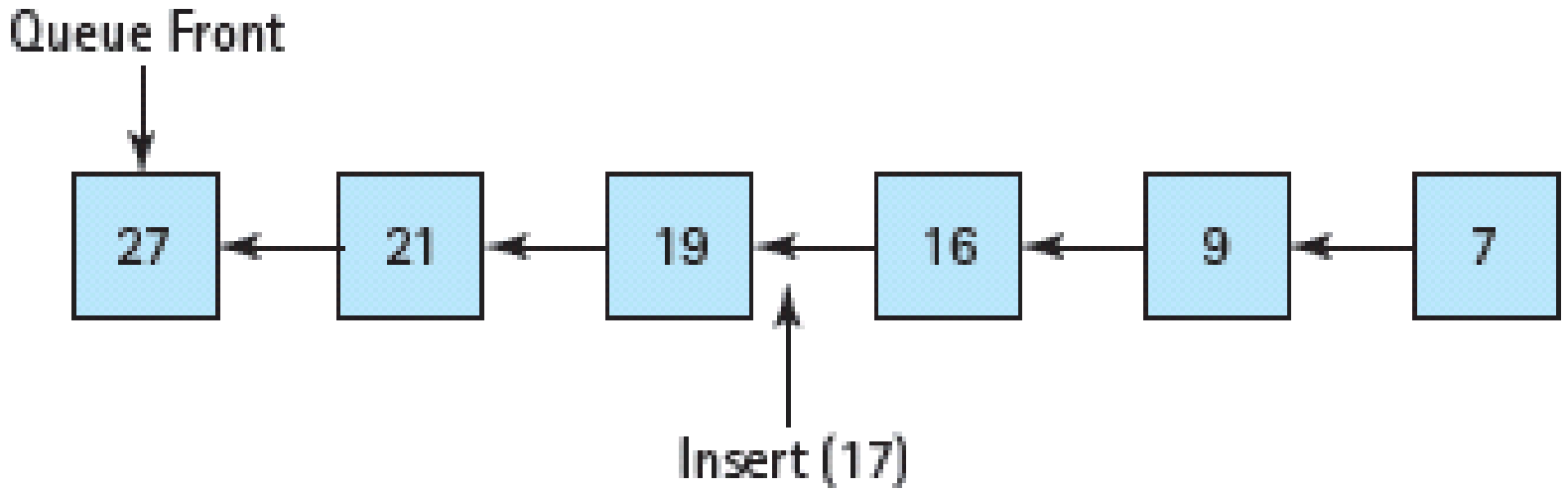
C++ Terminology for Queues

- C++ mostly uses stack terminology for queues: `push`, `pop`, `empty`, and `size`
- Consequently, one must be careful in naming stack and priority objects to distinguish what kind of object each one is
- Operation named, **`back`**, returns the element at the rear of the queue

Priority Queues

- A **priority queue** behaves like a queue in that data is removed only from its front
- Instead of inserting elements at the rear, however, elements are inserted so that they are in some prescribed order
- Example: Passengers boarding an airplane who are called to line up in order of their seat assignments

A Priority Queue, ADT Showing Insertion and Deletion Points



Bidirectional lists

Bidirectional lists appropriate to use when:

- one wants to make insertions and deletions in the middle of the data structures;
- random access is not as important as in array;
- the array is inappropriate either as an ADT or as an implementation;

Bidirectional lists, cont...

Bidirectional lists appropriate to use when:

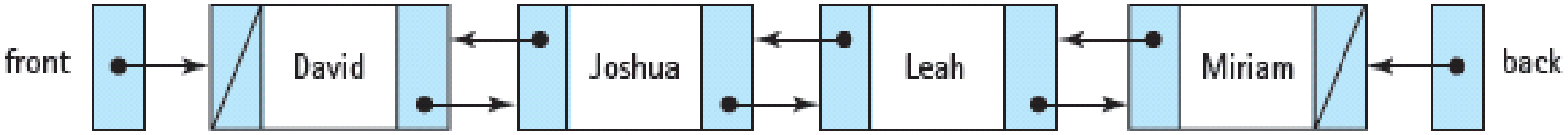
- a programmer wants a list structure that provides for bidirectional traversal;
- only in cases when a programmer needs these extra features because of its space inefficiency

Linked Implementation of the Bidirectional lists

In a linked implementation of the bidirectional list:

- List nodes augmented with a second link that points in the opposite directions
- External references kept to the front & the back of the list
- One has the ability to step through the nodes by going either forward or backward, See Figure 17.7

A Bidirectional Linked list



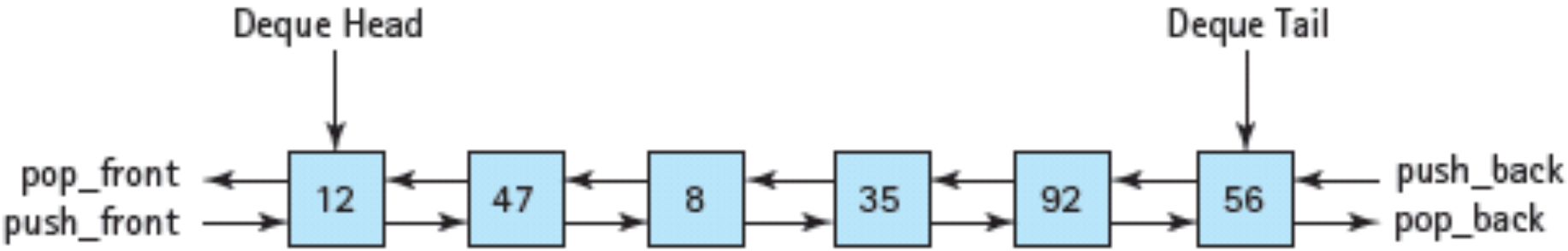
deque Operations

- C++ deque uses :
- `push_front` and `push_back` for insertion,
- `pop_front` and `pop_back` for deletion
- `front` and `back` for accessing the head an tail operations
- C++ deque is one huge class that combines a combination of the interfaces for the abstract directional list, the deque, and the array

deques

- A **deque** is the bidirectional equivalent of a queue
- With deques , a programmer can insert and delete items at either end
- Example: it could represent a train of railroad cars in a switchyard, where cars can be added and removed at either end of the train

Example of a deque



The `vector` Template

- The `vector` class implements or defines an array, and `vector` doesn't have to be fixed in size
- As a result, subarray processing becomes unnecessary as does the need to decide how big to make the array when one declares it
- `Vector must` be instantiated with a constructor at run time
- Thus, one cannot declare a `vector` using an initialization list as one can do with an array

Simple Calls to `Vector` Constructors

Examples:

```
vector<int> intVec(20); // A vector holding 20 integers
                        // all initialized to 0,
vector<string> noSizeVec; // A vector holding 0 strings
```

The first statement makes sense–similar to writing this code: `int intVec[20]`

But why declare a vector with no space in it, as in the second statement?

`vector` Constructors, cont. . .

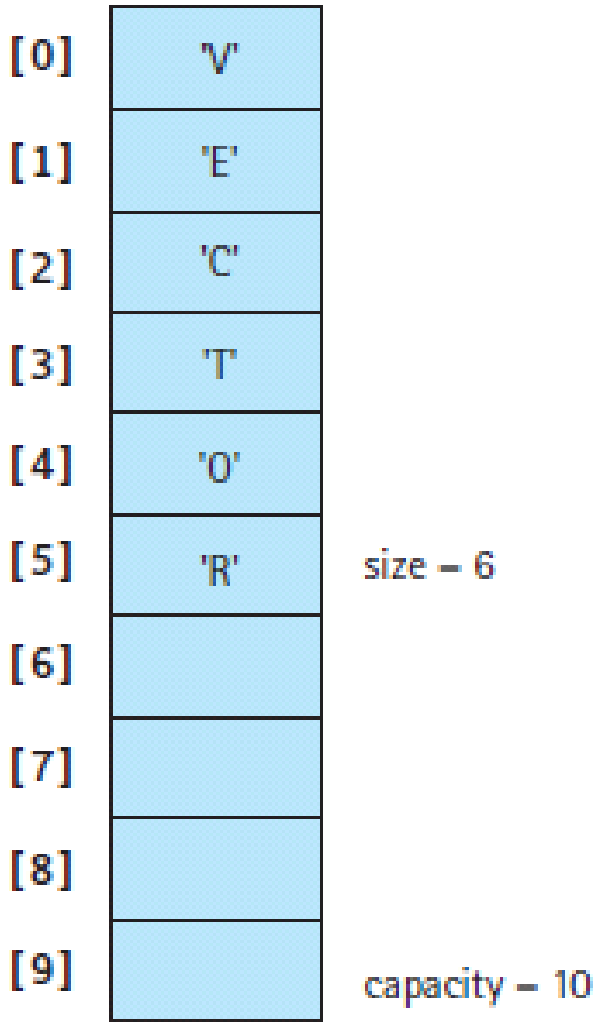
- The reason is that a vector automatically expands to hold as much data as one puts into it
- In the second statement , the programmer did not know how many strings he or she would put into `noSizeVec`
- Consequently, the programmer declares the vector with no elements and lets the structure grow as one inserts value

`vector` Constructors

- When one knows a maximum size in advance, one should indicate that value in the constructor
- Table on p. 886 of Ch. 17 summarizes operations of operators commonly used with `vector`

These operators include: `at`, `back`, `begin`, `capacity`, `clear`, `empty`, `end`, `erase`, `front`, `insert`, `push_back`, `rbegin`, `rend`, `reserve`, and `size`

A `vector <char>` for which 10 Spaces Reserved and 6 Elements Inserted



Initializing a `vector` from an Array of Values

While unable to use an initializer list with a **vector;** however, one can initialize it from an array of values

Example from program in Ch. 17 looking up number of occupants in a given apartment:

```
int occData[] =
  {3,4,0,3,4,1,1,2,3,2,1,4,2,3,0,5,2,3};
  vector<int> occupants(occData, occData
  +sizeof(occData)/sizeof(int));
```

Initializing a `vector` from an Array of Values, cont. . .

- First argument, `occData`, is the base address of the already initialized array
- Second argument, `occData+sizeof(occData)/sizeof(int)`, is the address that is one greater than the location of the last element of the array
- C++ automatically casts these addresses into iterators of the appropriate type as they are passed to the constructor
- Constructor designed so that the address just beyond the array doesn't copy that location into the `vector`

`list` Template

- `list` defines:

  `back`, `begin`, `clear`, `empty`, `end`, `erase`, `front`, `insert`, `push_back`, `rbegin`, `rend`, and `size` with similar meanings as for `vector`

Table on p.888 of ch.17 describes additional operators that `list` supports

- Because it is a linked implementation, `list` does not define a `capacity` or `reserve`

`list` Template

- Because it doesn't support random access, the `[]` operator is not overloaded nor is the `at` operation supported
- A `list` can be created:
  - with zero elements
  - with a specific number of elements initialized to a default value
  - by initializing from an array or another container
  - by using a copy constructor

Example: How a `List` Works

Following nonsense example program creates a list from user input and:

- displays it contents in different ways
- applies various mutator operations and
- merges it with another `list`

```
//*****************************************// This program demonstrates various list methods //
*******************************************
#include <iostream>
#include <list> using namespace std;
int main ()
{
  list<string> demo; // Create an empty list
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++) {
    cin >> word;
    demo.push_back(word); // Insert elements at back
  }
```

```
 // Access front, back, and size
  cout << "Front element: " << demo.front() << endl
 << "Back element: " << demo.back() << endl
 << "Size of list: " << demo.size() << endl;

// Create a forward iterator
  list<string>::iterator place;
 cout << "List contents from beginning to end: " <<
endl;

// Traverse list in forward direction
  for (place = demo.begin(); place != demo.end();
       place++)
```

```
cout << *rplace << " ";
cout << endl << "Enter a word to insert after the first
word: ";
cin >> word;
place = demo.begin(); // Point forward iterator to front +
+place; // Advance one place
demo.insert(place, word); // Insert an element
place = demo.end(); // Point forward iterator past end
--place; // Move back to last element
--place; // Move back one more place
demo.erase(place); // Delete element
cout << "Next to last word has been erased." << endl;
cout << "Enter a word to add at the front: ";
cin >> word;
demo.push_front(word); // Insert at front
cout << "List contents from beginning to end: " << endl;
```

The `stack` Template

- The goal of the `stack` type is to model a LIFO push-down stack as simply as possible
- Stack type has only five operators associated with it: `empty`, `top`, `push`, `pop`, and `size`
- See Table on p. 891 of Ch. 17 for detailed description of these operators' actions

Creating a `stack`

- Two ways to create a `stack`, either as an empty container or as copy of a `vector`, `list`, or `deque`
- Examples:

  1) `stack<float> fltStack;` **// Create an empty stack of floats stack<string>**

  2) `strStack(demo);` **// Create string stack from list demo**

The `queue` Template

- Simple design philosophy of `queue` very similar to `stack` except in a couple of respects:
- In `queue`, `push` inserts elements at one end of data structure and `pop` removes them from the other end.
- In addition to the observer, **`front`**, for the element that is about to be popped, **`queue`** provides **`back`** to allow one to look at element most recently pushed
- Table providing summary of `queue's` operations on p. 892 of Ch. 17

The `queue` Template

- Two forms of constructors for `queue`:
- 1) default creates an empty queue;
- 2) other form allows one to initialize the `queue` with a copy of the contents of `vector`, `list`, or `deque`

The `priority_queue` Template

- Major difference of `priority_queue` with `queue`: elements in the structure kept in sorted order with `priority_queue`
- Thus, `push` inserts elements using an insertion sort so that they go directly to their appropriate places in the ordering
- At one end, `top` observes the element with the greatest value and application of `pop` removes it
- Example of `priority_queue` : triage of patients in an emergency room of a hospital
- Table on p. 894 of ch.17 has a summary of its operations

The `deque` Template

- `deque` replicates most operations of `list` including: `back`, `begin`, `clear`, `empty`, `end`, `erase`, `front`, `insert`, `push_back`, `rbegin`, `rend`, and `size` with similar meanings
- As with `vector`, `deque` overloads `[]` and `at` to provide random access
- Being linked, it lacks a need to support `capacity` or `reserve` operations
- See Problem-Solving Case Study at end of Ch. 17 for example of use of `deque`

The `deque` Template

- `deque` does not directly support `merge`, `remove`, `reverse`, `sort`, or `unique`
- However, equivalent function templates with the same names, all applicable to `deque`, are defined in the algorithms section of the STL
- Increase in capability gained by using `deque` is offset by the costs in efficiency in running `deque`
- Thus, do not use `deque` unless you actually need the combined features of `list` and `vector`

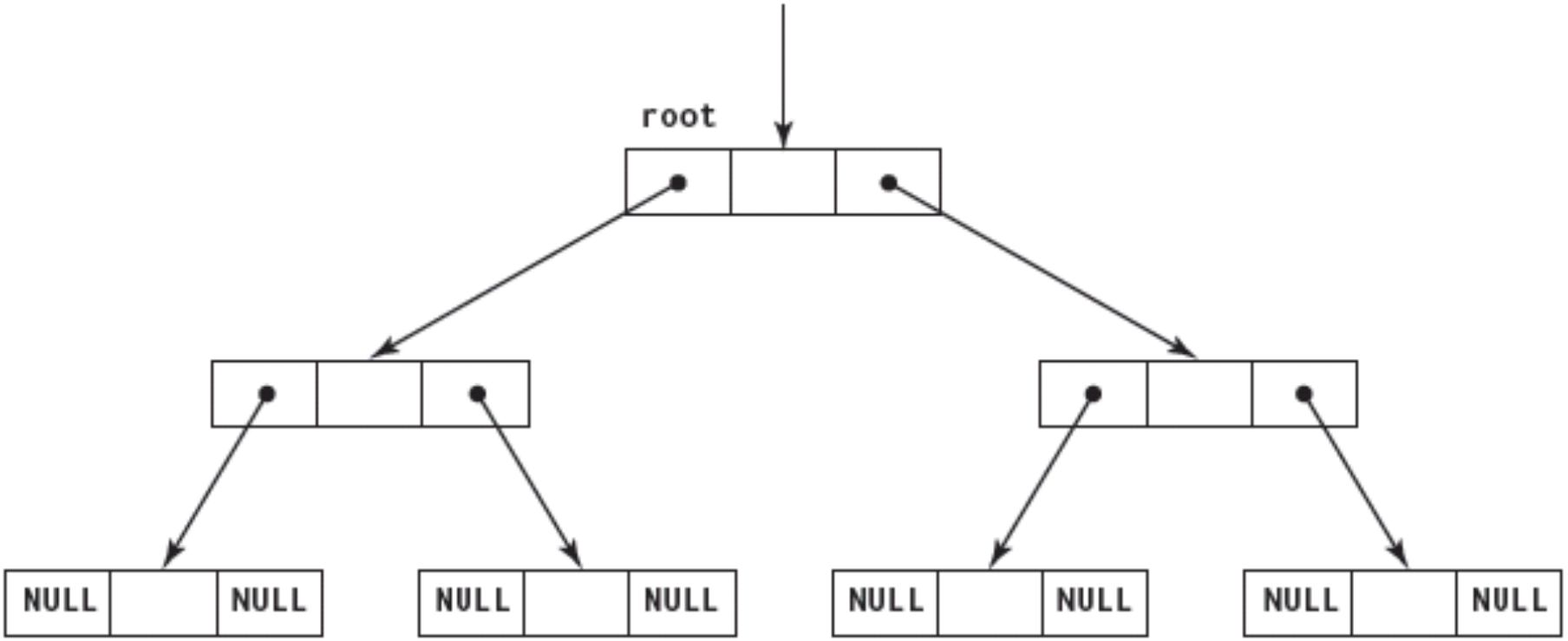
Nonlinear Structures: Binary Trees

- A binary tree is a nonlinear structure in which elements have links that branch out in just two different directions
- A binary tree is referenced by an external pointer to a specific node, called the *root* of the tree
- The root has two pointers: one to its *left child* and one to its *right child*
- The left and right children of a node are called *siblings*

Nonlinear Structures: Binary Trees

- For any node in a tree, the left child of the node is the root of the *left subtree* of the node
- Similarly, the right child is the root of the *right subtree*
- Nodes whose left and right children are both `NULL` are called *leaf nodes*
- No theoretical limit on the number of nodes in a tree

A Binary Tree



A Binary Search Tree

- A binary search tree– a special type of binary tree where the values are arranged in a particular order
- Component in any node is > component of its left child or any of its children (left subtree) and < the component of its right child or any of it children (right subtree)
- This above definition assumes no duplicates

Binary Search Tree

- Binary search trees useful in looking for a certain item because:
- one is able to identify which half of the tree is in with just one comparison and
- one is able then to tell which half of that half the item is in with one more comparison
- Process continues until either we find the item or we determine that the item is not there

Searching for # 50 in a Binary Search Tree

[Insert Illustration of Searching for # 50 on Binary Search Tree on Bottom of p. 909 in .pdf version]

Searching for #18, a Number not in Binary Search Tree

[Insert Binary Search Tree Diagram of for #18 on top of p.990 of .pdf version]

Traversing the Binary Search Tree

- Three commonly used traversal patterns for a tree:
  1) **in-order traversal**
  2) **pre-order traversal**
  3) **post-order traversal**

1) In-order traversal:
Visit the left subtree
Then, the visit root
Then, visit right subtree

2) Pre-order traversal:
Visit the root
Then, visit left subtree
Then, visit right subtree

Traversing the Binary Search Tree

3) Post-order traversal:

Visit the left sub-tree

Then, visit the right sub-tree

Then, Visit the root

- Depending on the data stored in a binary tree, each order of traversal may have a particular use

Example of In-order Traversal

[Copy and Paste Binary Child Binary Search Tree on Bottom of p. 910 of .pdf Ch.17]

Hash Functions

- A hash function is an algorithmic function that converts (hashes) large, usually different-sized amount of data into small datum
- This datum (hash value) is usually a single integer that may serve as an index into an array
- Hash functions mostly used to expedite table lookup or data comparison assignments such as locating items in a database or detecting duplicated or similar records in a data set

Hash Functions and Collisions

- A hash function may sort and store multiple values into the same place in the array, known as a **collision**
- Desirable to minimize collisions through different approaches such as **rehashing**
- Efficiency of hashing depends on having an array or table that is large enough to hold the data set
- Figure 17.12 shows a chained hash table, a well designed hash function with lists remaining fairly short and having few collisions

A Chained Hash Table

[Insert Figure 17.12 from top of p. 913 of pdf. version of text]

Associative Containers

- Associative containers permit one to locate values not by their positions, but by their associated values
- **Associative lookup** or **content addressable access** used for many kinds of data processing
- Value used for searching is called the **key**, and collectively these values must be ordered so that it allows two keys to be compared
- In many cases, the keys must also be unique, and `set` and `map` are two of STL's associative containers

The `set` Template

- The STL `set` template stored only unique values
- If one tries to insert a value already in a set, the set will be unchanged
- Internally, the values in a set are sorted from least to greatest to traverse the set as if it was a sorted list
- However, this is less efficient than traversing a sequence container such as `list`

`set` Operations

- `set` excels at finding elements by their values
- After `find` operator provides an iterator pointing to value in a `set`, one can use it to `erase` the element or retrieve adjacent elements
- Table on p. 915 of Ch. 17 contains description of `set` operations:

  `begin, clear, count empty, end, erase, find, insert, lower_bound , rbegin, rend, size, and upper_bound`

`set` Algorithms

- `set_difference, set_intersection, and set_ union` are the three `set` algorithms
- They are called like traditional functions and must include <algorithm> to use them
- All of the set algorithms use same arrangement of five parameters, (iterators):
  `set_difference(start1, end1, start2, end2, and output);`

Example of call to `set_difference`

- `Inserter` must be used to force the `set` algorithms to generate their output in a way that enables a `set` to insert the output into itself
- Example of a call to `set_difference`:

```
set_difference(set1.begin(), set1.end(), // First set
set2.begin(), set2.end(), // Second set
inserter(set3, set3.end())); // Result set
```

In this example, `set3` contains all of the elements of `set1` that are not found in `set2`

The `map` Template

- An STL `map` is like a `set` with a major difference:
- `set` stores only key values; `map` associates second value with each key
- `map` permits one to implement the kind of name-index array, such as a phone book illustrated in Figure 17.14
- Although conceptually an array, a `map` may be implemented internally with a linked structure such as binary search tree or a chained hash table

`map` Operations

- Elements of operations are sorted
- `map` supports operations `begin`, `clear`, `cout`, `empty`, `end`, `erase`, `find`, `insert`, `lower_bound`, `rbegin`, `rend`, `size`, and `upper_bound`
- `Insert` operation expects a value of the template type `pair`

A Conceptual Illustration of a `map` for a phone book

[Insert Figure 17.14 Top of p. 916]

Example of Using a `map` for a Phone Book

Here is an example definition of a **map** that could be used for a phone book, and an **insert** operation that places an entry into it:

```
map<string, int> phonebook;
phonebook.insert(pair<string, int>("Eben
Johnson", 5550001));
```

Another Difference Between `set` and `map`

- With `map`, the `[ ]` operator is overloaded, enabling one to write code using concise syntax

For example, the following intricate code statements:

```
map<string, int>::iterator place; place =
   phonebook.find("Eben Johnson"); cout << *place.first
   << ": " << *place.second << endl;
   phonebook.erase(place);
```

can instead be written concisely as the following:

```
map<string, int> phonebook;
   string name = "Eben Johnson";
   phonebook[name] = 5550001;
   cout << name << ": " << phonebook[name] << endl;
   phonebook.erase(name);
```