Chapter 10
Vectors

3rd Edition

Computing Fundamentals with C++

Rick Mercer

Franklin, Beedle & Associates

Goals

- Construct and use vector objects that store collections of any type
- Implement algorithms to process a collection of objects
- Use the sequential search algorithm to locate a specific element in a vector
- Pass vector objects to functions
- Sort vector elements
- Understand how to search using the classic sequential and binary search algorithms

class Vector

- Some objects store precisely one value
  - a double store one number
  - an `int` stores one integer
- Other objects store more than one (possibly dissimilar) values, for example:
  - `BankAccount` objects store a `string` and a `double`
- What does a `string` object store?

Recall string objects

- Any string object stores a collection of characters, more than one value
- Individual characters are referenced with [ ]

```
cout << name[0]; // reference 1st character
```

- This chapter introduces `vector` objects
  - Store a indexed collection of objects
  - Individual objects are accessed through subscripts `[ ]`

vectors are Generic

- This code declares a vector named **`x`** that has the capacity to store 100 numbers

```
vector<double> x(100); // All garbage values
x[0] = 1.5;
x[1] = 6.3;
cout << x[0] + x[1]; // 7.8
```

- We can have a vector of almost any class of object

```
vector <int> tests(100);
vector <string> names(20);
vector <Employee> employees(1000);
vector<vector<int> > table(12);
```

vector construction

`vector`<*class*> *identifier* `(` *capacity*, *initial-value* `) ;`

- *class* specifies the class of objects stored in the vector
- *identifier* is the name of the vector object
- *capacity* is an integer expression specifying the maximum number of objects that can be stored
- *initial-value* is the value of every element
- *initial value* is optional
- Need to

```
#include <vector> // For vector<type>
```

Example Constructions

- A vector that stores up to 8 numbers, which are all initialized to 0.0

```
vector <double> x(8, 0.0);
```

- A vector that stores 500 string objects:

```
vector <string> name(500);
```

- A vector that store 1,000 integers, which are all initialized to -1):

```
vector <int> test(1000, -1);
```

- A vector that stores up to 100 BankAccounts

```
vector <BankAccount> customer(100);
```

Accessing Individual Elements in the Collection

- Individual array elements are referenced through subscripts of this form:

  vector-name [*int-expression*]

  - *int-expression* is an integer that should be in the range of 0..*capacity*-1.

- Examples:

```
x[0]          // Pronounced  x sub 0
name[5]       // Pronounced name sub 5
test[99]      // Pronounced test sub 99
customer[12] // Pronounced customer sub 12
```

A Complete Program

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  int n = 5;
  vector <int> x(n, 0);
  x[0] = 1;                         // Assume input of
  cout << "Enter two integers: "; //    2  5
  cin >> x[1] >> x[2];
  x[3] = x[0] + x[2];
  x[4] = x[3] - 1;
  for(int j = 0; j < n; j++) {
    cout << x[j] << " ";
  }
  return 0;
}
```

```
Enter two integers: 2 5
1 2 5 6 5
```

Another view of the vector<int>

Individual Element	Value
`x[0]`	1
`x[1]`	2
`x[2]`	5
`x[3]`	6
`x[4]`	5

```
Enter two integers: 2 5
1 2 5 6 5
```

Vector Processing with a Determinate Loop

- The need often arises to access all meaningful elements

```
vector <double> test(100, -99.9);

// Initialize the first 24 elements
test[0] = 64;
test[1] = 82;
// . . . assume 21 additional assignments . . .
test[23] = 97;
int n = 24; // The first 24 elements are meaningful

// Sum the first n elements in test
double sum = 0.0;
for (int j = 0; j < n; j++) {
  sum += test[j];
}
```

Processing the First `n` Elements of a vector

- A vector often has capacity larger than need be
  - The previous example only used the first 24 of a potential 100 elements.
  - The textbook often uses `n` to represent the number of initialized and meaningful elements
  - The previous loop did not add `x[24]` nor `x[25]`, nor `x[99]` *all of which were -99.9*
- vectors can be sized at runtime *and even resized later*

vector processing in this text book

- Example `vector` processing you will see
  - displaying some or all vector elements
  - finding the sum, average, largest, ... of all vector elements
  - searching for a given value in the `vector`
  - arranging elements in a certain order
    - ordering elements from largest to smallest
    - or alphabetizing a vector of strings from smallest to largest

Out of Range Subscript Checking

- Most vector classes don't care if you use subscripts that are out of range

```
vector<string> name(1000);
name[-1] = "Subscript too low";
name[0] = "This should be the first name";
name[999] = "This is the last good subscript";
name[1000] = "Subscript too high";
```

- This could crash your computer instead! *segmentation or general protection faults*

Subscript Checking

- `vector` does not perform range checking with `[ ]`
- The programmer must be careful to avoid subscripts that are not in the range
- Both assignments below do not cause a runtime error
  - Instead they store the values in memory that belongs to someone else, there is no error or warning

```
int n = 5;
 vector <int> x(n, 0);
 x[-1] = 123; // Too low
 x[5] = 123;  // Too high
```

Subscript Checking

- `vector` has a member function `at(int)` that does perform range checking
- If the subscript is out of range, you get a runtime error
- Both assignments below would cause a runtime error

```
int n = 5;
vector <int> x(n, 0);
x.at(-1) = 123; // Too low
x.at(5) = 123;  // Too high
```

```
libc++abi.dylib: terminating with uncaught exception
of type std::out_of_range: vector
```

vector::capacity and vector::resize

- The proper capacity of a vector is usually an issue
- There are two useful functions to help

```
// Maximum number of elements to be stored
int vector::capacity()

// Change the capacity
void vector::resize(int newSize)
```

vector::capacity and vector::resize

```
#include <vector> // for the standard vector class
#include <iostream>

using namespace std;
int main() {
  vector <int> v1;      // v1 cannot store any elements
  vector <int> v2(5);
  cout << "v1 can hold " << v1.capacity() << endl;
  cout << "v2 can hold " << v2.capacity() << endl;

  v1.resize(22);
  cout << "v1 can now hold " << v1.capacity() << endl;
  return 0;
}
```

Output

```
v1 can hold 0
v2 can hold 5
v1 can now hold 22
```

What happens during a resize message?

- When a vector is resized
  - and the new size is bigger than the old size
    - the existing elements are intact
  - and the new size is smaller than the old size
    - the elements in the highest locations are truncated

Sequential Search

- We often need to search for data stored in a vector (a phone number, an inventory item, an airline reservation, a bank customer)
- We will simplify the search algorithm by searching only for strings
- Imagine however that the vector may be a collection of bankAccounts, students, inventory, sales, employees, or reservations

Sequential search algorithm

- There are many searching algorithms
- We will study the *sequential search* algorithm with a simple collection of strings
- Here is the first cut at the algorithm:

  Initialize a vector of strings (call it friends)

  Get the name to search for (call it searchName)

  Try to find searchName

  Report on success or failure of search

The array being searched

- We'll use this data in our searches:

```
vector<string> friends(10);
int n = 4; // Number of meaningful elements
friends[0] = "Casey";
myFriends[1] = "Dylan";
friends[2] = "Jordan";
myFriends[3] = "Kelly";
```

- Note: We often have unused elements in a vector
- For example, we could add 6 more strings to the collection named `friends`

The Possibilities?

- `searchName` is in the vector
- `searchName` is *not* in the vector
- Complete this problem as a free function

```
int indexOf(string searchName,
            const vector<string> & names,
            int n)
```

- Calls look like this, expected returns in comments

```
indexOf( "Not Here", friends, n) // -1
indexOf( "Jordan", friends, n) // 2
```

Sequential Search

- This algorithm is called sequential search because it looks at each vector element from index 0 to index n-1 in sequence
- If searchName is found, return the index
- If the loop terminates with no find, return -1

```
int indexOf(string search,
            const vector<string> & names, int n) {
  for (int index = 0; index < n; index++) {
    if (names[index] == search)
    return index;
  }
  return -1; // search not in the vector
}
```

Trace `indexOf` for "Jordan"

- At index 2, `indexOf` returns 2 when the `if` statements is true

Loop Iteration	`searchName`	`n`	`if`	`index`	Vector element
before	`"Jordan"`	`4`	`N/A`	`N/A`	`N/A`
#1	"	"	`false`	`0`	`"Casey"`
#2	"	"	`false`	`1`	`"Dylan"`
#3	"	"	`true`	`2`	`"Jordan"`

Trace `indexOf` when not found

- The loop terminates when index goes from 3 to 4
- `indexOf` then returns -1

Loop Iteration	**searchName**	**n**	**if**	**index**	Vector element
before	"Not Here"	4	N/A	N/A	N/A
#1	"	"	false	0	"Casey"
#2	"	"	false	1	"Dylan"
#3	"	"	false	2	"Jordan"
#4	"	"	false	3	"Kelly

Messages to individual objects

- General form for sending a message to an individual object in a vector:

  vector-name [*subscript*] . *message*

- Examples:

```
vector<string> name(1000);
vector<BankAccount> acct(10000);

acct[0] = BankAccount("Kelsey", 0.0);
acct[0].deposit(20.00);
acct[0].withdraw(10.00);
cout << acct[0].getBalance() << endl;
cout << acct[0].getName() << endl;
```

Initializing a vector with File Input

- A vector is often initialized with file input
- For example, might need to initialize a data base of bank customers with this file input:

```
Cust0            0.00
AnyName        111.11
Austen         222.22
Chelsea        333.33
Kieran         444.44
Cust5          555.55
```

... Seven lines are omitted ...

```
Cust11        1111.11
```

Some preliminaries

```
// Initialize a vector of BankAccounts with file input
#include <istream>    // for class ifstream
#include <iostream>   // for cout
#include <vector>     // for the standard vector class
#include "BankAccount.h"  // for class BankAccount
using namespace std;
int main() {
  ifstream inFile("bank.data");
  if (!inFile){
    cout << "*Error* 'bank.data' not found" << endl;
  } else {

   // . . . Read all lines from bank.data . . .
```

Reading until end of file

```
vector<BankAccount> account(20);
string name;
double balance = 0.0;
int n = 0;

while ((inFile >> name >> balance) && (n <
account.capacity())) {
  // Create and store a new BankAccount
  account[n] = BankAccount(name, balance);
  // Increase total of the accounts on file and
  // get ready to locate the next new BankAccount
  n++;
}
```

vector Argument/Parameter Associations

by example

```
void foo(vector<BankAccount> accounts) {
  // VALUE parameter  (should not be used with vectors)
  // all elements of accounts are copied
  // after allocating the additional memory
}

void foo(vector<BankAccount> & accounts) {
  // REFERENCE parameter  (allows changes to argument)
  // Only a pointer the accounts is copied.
  // A change to accounts changes the argument
}

void foo(const vector<BankAccount> & accounts) {
  // CONST REFERENCE parameter  (for efficiency and safety)
  // Only a reference to the accounts is copied (4 bytes)
  // A change to accounts does NOT change the argument
}
```

Sorting

- *Sorting:* the process of arranging vector elements into ascending or descending order
- Natural, or ascending order, where x is a vector object

```
x[0] <= x[1] <= x[2] <= ... <= x[n-2] <= x[n-1]
```

- Here's the data used in the next few slides:

Element	Unsorted	Sorted
data[0]	76.0	63.0
data[1]	74.0	74.0
data[2]	100.0	76.0
data[3]	62.0	89.0
data[4]	89.0	100.0

Swap smallest into index 0

```
// Find the index of the smallest element
left= 0
indexOfSmallest = left
for  index ranging from left+1 through n – 1 {
   if data[index ] < data[ indexOfSmallest ] then
       indexOfSmallest = index
}
// Question: What is smallestIndex now? ___________
swap data[ smallestIndex ] with data[ top ]
```

Selection sort algorithm

- Now we can sort the entire vector by changing left from 0 to n-2 with this loop

  for (left = 0; left < n-1; left++)

    for each subvector, get the smallest to data[left] (algorithm on previous slide)

- The index moves up one index vector position each time the element at the indexOfSmallest is swapped to the index
  - It is certainly possible the data[indexOfSmallest] is data[left]

Selection Sort

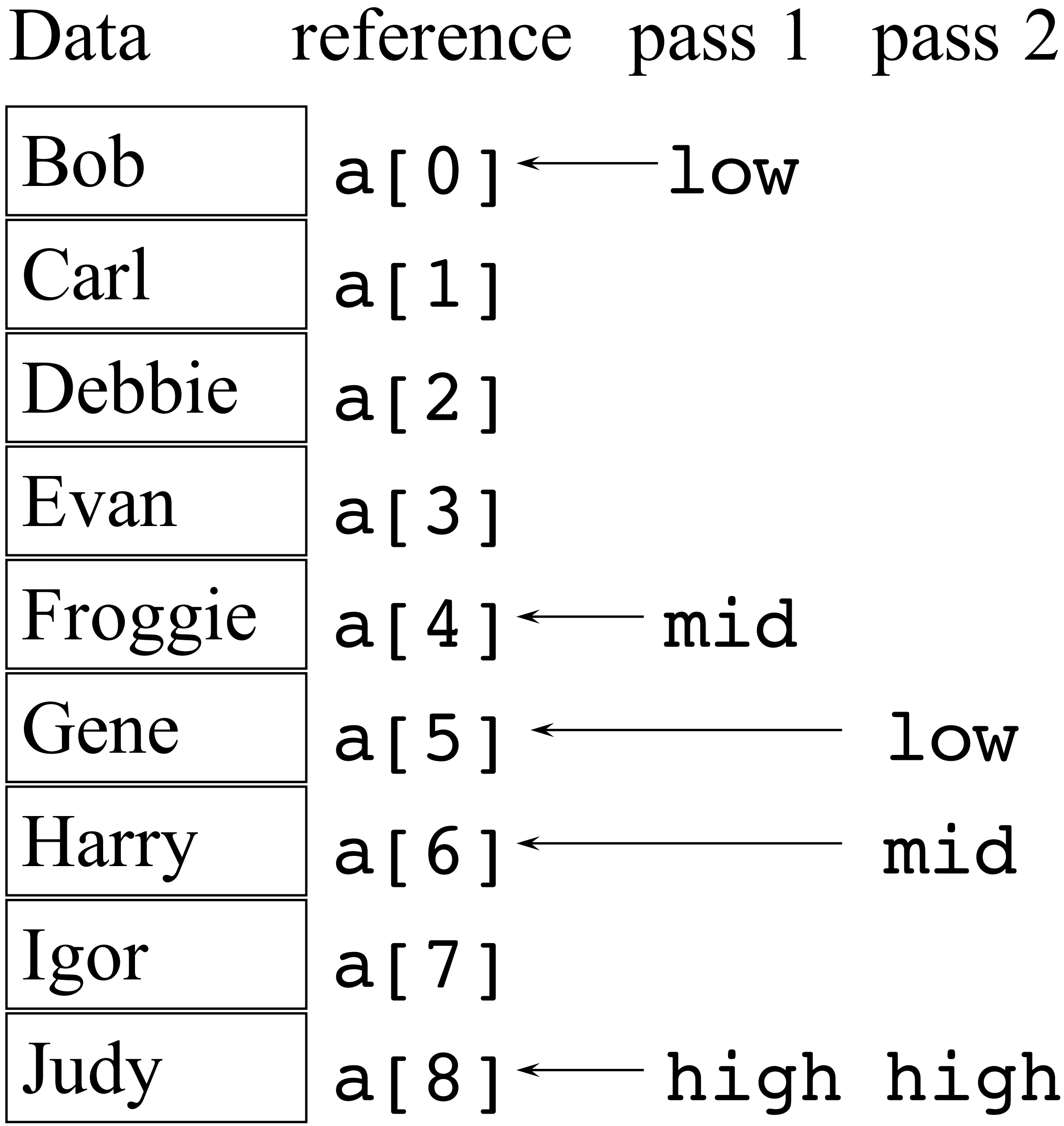
- This swap occurs when left is 0
    - 62 is swapped with `data[left]` when `left == 0`

`top == 0`	**Before**	**After**
`data[0]`	**76.0**	**62.0**
`data[1]`	91.0	91.0
`data[2]`	100.0	100.0
`data[3]`	**62.0**	**76.0**
`data[4]`	89.0	89.0

- With `left++`, `76.0` will be swapped with `91.0`

Binary Search

- We'll see that binary search can be a more efficient algorithm for searching
  - It works only on sorted arrays like this
    - Compare the element in the middle
    - if that's the target, quit and report success
    - if the key is smaller, search the array to the left
    - otherwise search the array to the right
  - This process repeats until we find the target or there is nothing left to search



Data
reference
pass 1
pass 2
Bob
Carl
Debbie
Evan
Froggie
Gene
Harry
Igor
Judy
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
low
mid
low
mid
high
high

How fast is Binary Search?

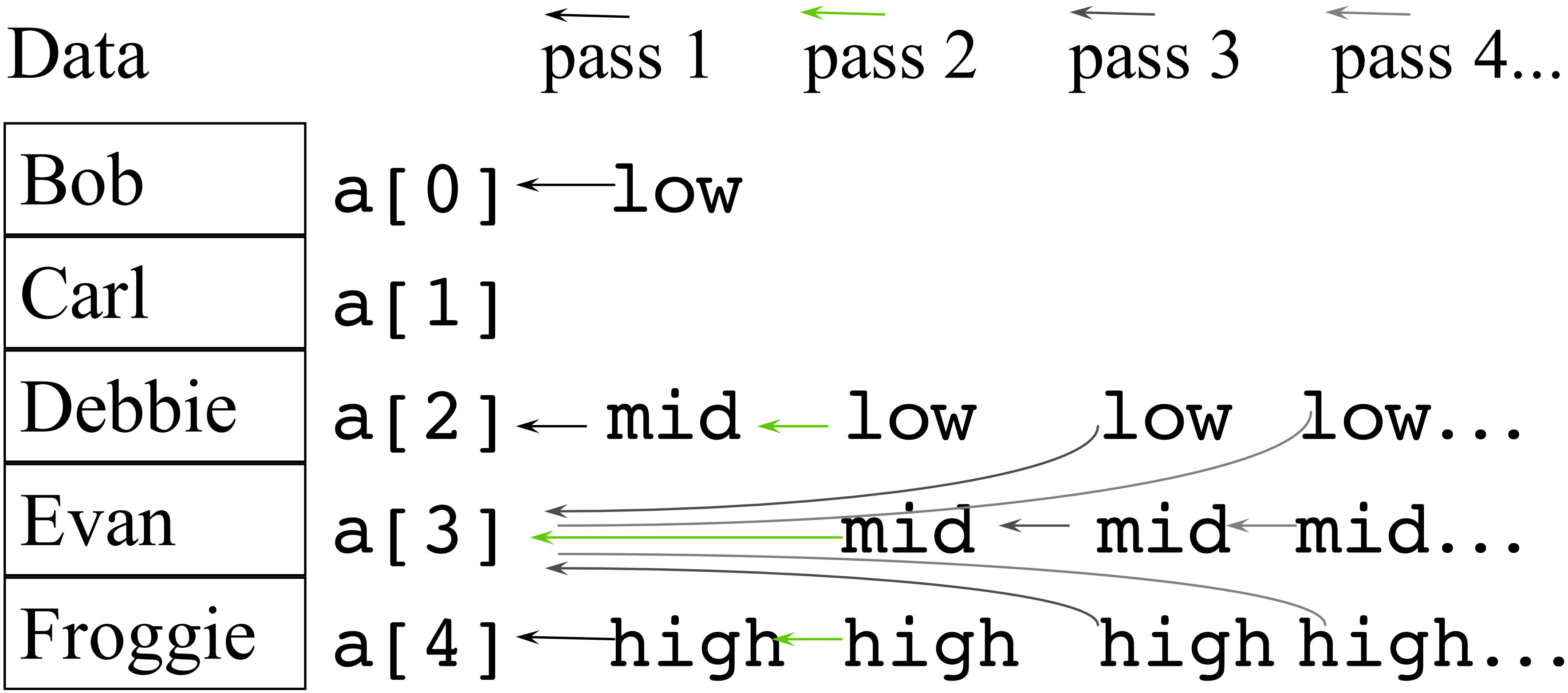
- Best case: 1 comparison
- Worst case: when the target is not there
- At each pass, the live portion of the array (where we need to search) is narrowed to half the previous size
- The series proceeds like this:
  - n , n/2, n/4, n/8, ...
- Each term in the series represents one comparison How long does it take to get to 1?
  - This will be the number of comparisons

Defective Binary Search

- Binary search sounds simple, but it's tricky *consider this code*

```
int binarySearch(const vector<int> & a, int n, int target) {
  // pre: array a is sorted from a[0] to a[n-1]
  int first = 0;
  int last = n - 1;
  int mid;
  while (first <= last) {
    mid = (first + last) / 2;
    if (target == a[mid])
      return mid;  // found target
    else if (target < a[mid])
      last = mid;  // must be that target > a[mid]
    else
      first = mid; // must be that target > a[mid]
  }
  return -1;       // use -1 to indicate item not found
}
```

It's an Infinite Loop



- How do we fix this defective binary search ?