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
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 [ ]
  ```cpp
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 \texttt{x} that has the capacity to store 100 numbers

\begin{verbatim}
vector<double> x(100); // All garbage values
x[0] = 1.5;
x[1] = 6.3;
cout << x[0] + x[1]; // 7.8
\end{verbatim}

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

\begin{verbatim}
vector <int> tests(100);
vector <string> names(20);
vector <Employee> employees(1000);
vector<vector<int>> table(12);
\end{verbatim}
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

```cpp
#include <vector>  // For vector<typename>
```
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:
  \[ \text{vector-name} [ \text{int-expression} ] \]
  • \textit{int-expression} is an integer that should be in the range of 0..\textit{capacity}-1.

• Examples:
  
  \begin{verbatim}
  x[0]     // Pronounced x sub 0
  name[5]  // Pronounced name sub 5
  test[99] // Pronounced test sub 99
  customer[12] // Pronounced customer sub 12
  \end{verbatim}
A Complete Program

```cpp
#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];
    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>

<table>
<thead>
<tr>
<th>Individual Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x[0]}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{x[1]}</td>
<td>2</td>
</tr>
<tr>
<td>\texttt{x[2]}</td>
<td>5</td>
</tr>
<tr>
<td>\texttt{x[3]}</td>
<td>6</td>
</tr>
<tr>
<td>\texttt{x[4]}</td>
<td>5</td>
</tr>
</tbody>
</table>

Enter two integers: \texttt{2 5}
\texttt{1 2 5 6 5}
Vector Processing with a Determinate Loop

• The need often arises to access all meaningful elements

```cpp
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

```cpp
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.

```cpp
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

```cpp
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
```
The proper capacity of a vector is usually an issue.

There are two useful functions to help:

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

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

  ```cpp
  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
  ```cpp
  int indexOf(string searchName,
              const vector<string> & names,
              int n)
  ```
- Calls look like this, expected returns in comments
  ```cpp
  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

```cpp
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

<table>
<thead>
<tr>
<th>Loop Iteration</th>
<th>searchName</th>
<th>n</th>
<th>if</th>
<th>index</th>
<th>Vector element</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>&quot;Jordan&quot;</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>#1</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>0</td>
<td>&quot;Casey&quot;</td>
</tr>
<tr>
<td>#2</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>1</td>
<td>&quot;Dylan&quot;</td>
</tr>
<tr>
<td>#3</td>
<td>&quot;</td>
<td>&quot;</td>
<td>true</td>
<td>2</td>
<td>&quot;Jordan&quot;</td>
</tr>
</tbody>
</table>
Trace `indexOf` when not found

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

<table>
<thead>
<tr>
<th>Loop Iteration</th>
<th>searchName</th>
<th>n</th>
<th>if</th>
<th>index</th>
<th>Vector element</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>&quot;Not Here&quot;</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>#1</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>0</td>
<td>&quot;Casey&quot;</td>
</tr>
<tr>
<td>#2</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>1</td>
<td>&quot;Dylan&quot;</td>
</tr>
<tr>
<td>#3</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>2</td>
<td>&quot;Jordan&quot;</td>
</tr>
<tr>
<td>#4</td>
<td>&quot;</td>
<td>&quot;</td>
<td>false</td>
<td>3</td>
<td>&quot;Kelly&quot;</td>
</tr>
</tbody>
</table>
Messages to individual objects

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

   \textit{vector-name [ subscript ] \textbullet message}

• Examples:

   \texttt{vector\langle string\rangle name(1000);}
   \texttt{vector\langle BankAccount\rangle acct(10000);}

   \texttt{acct[0] = BankAccount("Kelsey", 0.0);}
   \texttt{acct[0].deposit(20.00);}
   \texttt{acct[0].withdraw(10.00);}
   \texttt{cout \ll acct[0].getBalance() \ll endl;}
   \texttt{cout \ll acct[0].getName() \ll endl;}
Initializing a vector with File Input

• A vector is often initialized with file input

• For example, might need to initialize a database 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 . . .
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
  $x[0] \leq x[1] \leq x[2] \leq \ldots \leq x[n-2] \leq x[n-1]$
- Here's the data used in the next few slides:

<table>
<thead>
<tr>
<th>Element</th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[0]</td>
<td>76.0</td>
<td>63.0</td>
</tr>
<tr>
<td>data[1]</td>
<td>74.0</td>
<td>74.0</td>
</tr>
<tr>
<td>data[2]</td>
<td>100.0</td>
<td>76.0</td>
</tr>
<tr>
<td>data[3]</td>
<td>62.0</td>
<td>89.0</td>
</tr>
<tr>
<td>data[4]</td>
<td>89.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>top == 0</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[0]</td>
<td>76.0</td>
<td>62.0</td>
</tr>
<tr>
<td>data[1]</td>
<td>91.0</td>
<td>91.0</td>
</tr>
<tr>
<td>data[2]</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>data[3]</td>
<td>62.0</td>
<td>76.0</td>
</tr>
<tr>
<td>data[4]</td>
<td>89.0</td>
<td>89.0</td>
</tr>
</tbody>
</table>

- 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
<table>
<thead>
<tr>
<th>Data</th>
<th>reference</th>
<th>pass 1</th>
<th>pass 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>a[0] ← low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carl</td>
<td>a[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debbie</td>
<td>a[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evan</td>
<td>a[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Froggie</td>
<td>a[4] ← mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gene</td>
<td>a[5] ← low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harry</td>
<td>a[6] ← mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Igor</td>
<td>a[7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Judy</td>
<td>a[8] ← high high</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

```cpp
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

| Data | Bob     | a[0] ← low |
|      | Carl    | a[1]     |
|      | Debbie  | a[2] ← mid ← low ← low ← low... |
|      | Evan    | a[3] ← mid ← mid ← mid... |
|      | Froggie | a[4] ← high ← high ← high... |

- How do we fix this defective binary search?