Chapter 7, Arrays and Vectors

C++ How to Program, 8/e

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array's elements to zeros and printing the array.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int n[ 10 ]; // n is an array of 10 integers
10
11    // initialize elements of array n to 0
12    for ( int i = 0; i < 10; ++i )
13       n[ i ] = 0; // set element at location i to 0
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
16
17    // output each array element's value
18    for ( int j = 0; j < 10; ++j )
19       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
20 } // end main
```

Fig. 7.3 | Initializing an array's elements to zeros and printing the array. (Part 1 of 2.)

```
1  // Fig. 7.4: fig07_04.cpp
2  // Initializing an array in a declaration.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // use initializer list to initialize array n
10    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12    cout << "Element" << setw( 13 ) << "Value" << endl;
13
14    // output each array element's value
15    for ( int i = 0; i < 10; ++i )
16       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17 } // end main
```

Fig. 7.4 | Initializing an array in a declaration. (Part 1 of 2.)

7.4.2 Initializing an Array in a Declaration with an Initializer List

- The elements of an array also can be initialized in the array declaration by following the array name with an equals sign and a brace-delimited comma-separated list of initializers.
- The program in Fig. 7.4 uses an initializer list to initialize an integer array with 10 values (line 10) and prints the array in tabular format (lines 12–16).
- If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero.
- If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list.

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 20.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // constant variable can be used to specify array size
10    const int arraySize = 10;
11
12    int s[ arraySize ]; // array s has 10 elements
13
14    for ( int i = 0; i < arraySize; ++i ) // set the values
15       s[ i ] = 2 + 2 * i;
16
17    cout << "Element" << setw( 13 ) << "Value" << endl;
18
19    // output contents of array s in tabular format
20    for ( int j = 0; j < arraySize; ++j )
21       cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
22 } // end main
```

Fig. 7.5 | Generating values to be placed into elements of an array. (Part 1 of 2.)

Common Programming Error 7.2

Not initializing a constant variable when it's declared is a compilation error.

Common Programming Error 7.3

Assigning a value to a constant variable in an executable statement is a compilation error.

```
1  // Fig. 7.6: fig07_06.cpp
2  // Using a properly initialized constant variable.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int x = 7; // initialized constant variable
9
10    cout << "The value of constant variable x is: " << x << endl;
11 } // end main
```

```
The value of constant variable x is: 7
```

Fig. 7.6 | Using a properly initialized constant variable.

Common Programming Error 7.4

Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.

```
1  // Fig. 7.8: fig07_08.cpp
2  // Compute the sum of the elements of the array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int arraySize = 10; // constant variable indicating size of array
9     int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10    int total = 0;
11
12    // sum contents of array a
13    for ( int i = 0; i < arraySize; ++i )
14       total += a[ i ];
15
16    cout << "Total of array elements: " << total << endl;
17 } // end main
```

```
Total of array elements: 849
```

Fig. 7.8 | Computing the sum of the elements of an array.

```
1  // Fig. 7.11: fig07_11.cpp
2  // Poll analysis program.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // define array sizes
10    const int responseSize = 20; // size of array responses
11    const int frequencySize = 6; // size of array frequency
12
13    // place survey responses in array responses
14    const int responses[ responseSize ] = { 1, 2, 5, 4, 3, 5, 2, 1, 3,
15       1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5 };
16
17    // initialize frequency counters to 0
18    int frequency[ frequencySize ] = {};
19
```

Fig. 7.11 | Poll analysis program. (Part 1 of 2.)

```
20      // for each answer, select responses element and use that value
21      // as frequency subscript to determine element to increment
22      for ( int answer = 0; answer < responseSize; ++answer )
23         ++frequency[ responses[ answer ] ];
24
25      cout << "Rating" << setw( 17 ) << "Frequency" << endl;
26
27      // output each array element's value
28      for ( int rating = 1; rating < frequencySize; ++rating )
29         cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
30            << endl;
31   } // end main
```

```
Rating        Frequency
     1                3
     2                5
     3                7
     4                2
     5                3
```

Fig. 7.11 | Poll analysis program. (Part 2 of 2.)

Common Programming Error 7.5

Referring to an element outside the array bounds is an execution-time logic error. It isn't a syntax error.

7.4.8 Static Local Arrays and Automatic Local Arrays

- A program initializes `static` local arrays when their declarations are first encountered.
- If a `static` array is not initialized explicitly by you, each element of that array is initialized to zero by the compiler when the array is created.

Performance Tip 7.1

We can apply `static` to a local array declaration so that it not created and initialized each time the program calls the function and is not destroyed each time the function terminates. This can improve performance, especially when using large arrays.

```
1  // Fig. 7.12: fig07_12.cpp
2  // Static arrays are initialized to zero.
3  #include <iostream>
4  using namespace std;
5
6  void staticArrayInit( void ); // function prototype
7  void automaticArrayInit( void ); // function prototype
8  const int arraySize = 3;
9
10 int main()
11 {
12    cout << "First call to each function:\n";
13    staticArrayInit();
14    automaticArrayInit();
15
16    cout << "\n\nSecond call to each function:\n";
17    staticArrayInit();
18    automaticArrayInit();
19    cout << endl;
20 } // end main
21
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 1 of 4.)

```
22  // function to demonstrate a static local array
23  void staticArrayInit( void )
24  {
25     // initializes elements to 0 first time function is called
26     static int array1[ arraySize ]; // static local array
27
28     cout << "\nValues on entering staticArrayInit:\n";
29
30     // output contents of array1
31     for ( int i = 0; i < arraySize; ++i )
32        cout << "array1[" << i << "] = " << array1[ i ] << "   ";
33
34     cout << "\nValues on exiting staticArrayInit:\n";
35
36     // modify and output contents of array1
37     for ( int j = 0; j < arraySize; ++j )
38        cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "   ";
39  } // end function staticArrayInit
40
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 2 of 4.)

```
41  // function to demonstrate an automatic local array
42  void automaticArrayInit( void )
43  {
44     // initializes elements each time function is called
45     int array2[ arraySize ] = { 1, 2, 3 }; // automatic local array
46
47     cout << "\n\nValues on entering automaticArrayInit:\n";
48
49     // output contents of array2
50     for ( int i = 0; i < arraySize; ++i )
51        cout << "array2[" << i << "] = " << array2[ i ] << "   ";
52
53     cout << "\nValues on exiting automaticArrayInit:\n";
54
55     // modify and output contents of array2
56     for ( int j = 0; j < arraySize; ++j )
57        cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "   ";
58  } // end function automaticArrayInit
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 3 of 4.)

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 4 of 4.)

7.5 Passing Arrays to Functions

- To pass an array argument to a function,
  - specify the name of the array without any brackets
  - array size is normally passed as well
  - the called functions can modify the element values in the callers' original arrays

- The name of the array evaluates to:
  - the address of the first element of the array.
  - so, when you pass the name of the array, you're really passing a copy of the address of the array, which lets the function access the original

7.5 Passing Arrays to Functions (cont.)

- When you pass an individual array element, you pass a copy of the value of that element
  - use the subscripted name of the array element as an argument in the function call.

```
1  // Fig. 7.13: fig07_13.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void modifyArray( int [], int ); // appears strange; array and size
8  void modifyElement( int ); // receive array element value
9
10 int main()
11 {
12    const int arraySize = 5; // size of array a
13    int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a
14
15    cout << "Effects of passing entire array by reference:"
16       << "\nThe values of the original array are:\n";
17
18    // output original array elements
19    for ( int i = 0; i < arraySize; ++i )
20       cout << setw( 3 ) << a[ i ];
21
22    cout << endl;
23
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 1 of 4.)

```
24      // pass array a to modifyArray by reference
25      modifyArray( a, arraySize );
26      cout << "The values of the modified array are:\n";
27
28      // output modified array elements
29      for ( int j = 0; j < arraySize; ++j )
30         cout << setw( 3 ) << a[ j ];
31
32      cout << "\n\nEffects of passing array element by value:"
33         << "\na[3] before modifyElement: " << a[ 3 ] << endl;
34
35      modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
36      cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
37   } // end main
38
39   // in function modifyArray, "b" points to the original array "a" in memory
40   void modifyArray( int b[], int sizeOfArray )
41   {
42      // multiply each array element by 2
43      for ( int k = 0; k < sizeOfArray; ++k )
44         b[ k ] *= 2;
45   } // end function modifyArray
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 2 of 4.)

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 3 of 4.)

```
47  // in function modifyElement, "e" is a local copy of
48  // array element a[ 3 ] passed from main
49  void modifyElement( int e )
50  {
51     // multiply parameter by 2
52     cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
53  } // end function modifyElement
```

```
Effects of passing entire array by reference:
The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element by value:
a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 4 of 4.)

Software Engineering Observation 7.3

Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it's absolutely necessary.

```
1  // Fig. 7.14: fig07_14.cpp
2  // Demonstrating the const type qualifier.
3  #include <iostream>
4  using namespace std;
5
6  void tryToModifyArray( const int [] ); // function prototype
7
8  int main()
9  {
10    int a[] = { 10, 20, 30 };
11
12    tryToModifyArray( a );
13    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
14 } // end main
15
16 // In function tryToModifyArray, "b" cannot be used
17 // to modify the original array "a" in main.
18 void tryToModifyArray( const int b[] )
19 {
20    b[ 0 ] /= 2; // compilation error
21 } // end function tryToModifyArray
```

Fig. 7.14 | const type qualifier applied to an array parameter. (Part 1 of 2.)

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch07\fig07_14\fig07_14.cpp(20) : error C3892: 'b' : you
   cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_14.cpp:20: error: assignment of read-only location
```

Fig. 7.14 | `const` type qualifier applied to an array parameter. (Part 2 of 2.)

7.6 Case Study: Class GradeBook Using an Array to Store Grades

- This section further evolves class `GradeBook`, introduced in Chapter 3 and expanded in Chapters 4–6.
- Previous versions of the class process grades entered by the user, but do not maintain the individual grade values in the class's data members.
- Thus, repeat calculations require the user to reenter the grades.
- In this section, we store grades in an array.

```
1  // Fig. 7.15: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ Standard Library string class
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    // constant -- number of students who took the test
12    static const int students = 10; // note public data
13
14    // constructor initializes course name and array of grades
15    GradeBook( string, const int [] );
16
```

Fig. 7.15 | Definition of class GradeBook that uses an array to store test grades. (Part 1 of 2.)

```
17      void setCourseName( string ); // function to set the course name
18      string getCourseName(); // function to retrieve the course name
19      void displayMessage(); // display a welcome message
20      void processGrades(); // perform various operations on the grade data
21      int getMinimum(); // find the minimum grade for the test
22      int getMaximum(); // find the maximum grade for the test
23      double getAverage(); // determine the average grade for the test
24      void outputBarChart(); // output bar chart of grade distribution
25      void outputGrades(); // output the contents of the grades array
26   private:
27      string courseName; // course name for this grade book
28      int grades[ students ]; // array of student grades
29   }; // end class GradeBook
```

Fig. 7.15 | Definition of class GradeBook that uses an array to store test grades. (Part 2 of 2.)

```
1  // Fig. 7.16: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses an array to store test grades.
4  #include <iostream>
5  #include <iomanip>
6  #include "GradeBook.h" // GradeBook class definition
7  using namespace std;
8
9  // constructor initializes courseName and grades array
10 GradeBook::GradeBook( string name, const int gradesArray[] )
11 {
12    setCourseName( name ); // initialize courseName
13
14    // copy grades from gradesArray to grades data member
15    for ( int grade = 0; grade < students; ++grade )
16       grades[ grade ] = gradesArray[ grade ];
17 } // end GradeBook constructor
18
19 // function to set the course name
20 void GradeBook::setCourseName( string name )
21 {
22    courseName = name; // store the course name
23 } // end function setCourseName
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 1 of 8.)

```
24
25  // function to retrieve the course name
26  string GradeBook::getCourseName()
27  {
28     return courseName;
29  } // end function getCourseName
30
31  // display a welcome message to the GradeBook user
32  void GradeBook::displayMessage()
33  {
34     // this statement calls getCourseName to get the
35     // name of the course this GradeBook represents
36     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
37        << endl;
38  } // end function displayMessage
39
40  // perform various operations on the data
41  void GradeBook::processGrades()
42  {
43     outputGrades(); // output grades array
44
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 2 of 8.)

```
45      // display average of all grades and minimum and maximum grades
46      cout << "\nClass average is " << setprecision( 2 ) << fixed <<
47         getAverage() << "\nLowest grade is " << getMinimum() <<
48         "\nHighest grade is " << getMaximum() << endl;
49
50      outputBarChart(); // print grade distribution chart
51   } // end function processGrades
52
53   // find minimum grade
54   int GradeBook::getMinimum()
55   {
56      int lowGrade = 100; // assume lowest grade is 100
57
58      // loop through grades array
59      for ( int grade = 0; grade < students; ++grade )
60      {
61         // if current grade lower than lowGrade, assign it to lowGrade
62         if ( grades[ grade ] < lowGrade )
63            lowGrade = grades[ grade ]; // new lowest grade
64      } // end for
65
66      return lowGrade; // return lowest grade
67   } // end function getMinimum
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 3 of 8.)

```
68
69  // find maximum grade
70  int GradeBook::getMaximum()
71  {
72     int highGrade = 0; // assume highest grade is 0
73
74     // loop through grades array
75     for ( int grade = 0; grade < students; ++grade )
76     {
77        // if current grade higher than highGrade, assign it to highGrade
78        if ( grades[ grade ] > highGrade )
79           highGrade = grades[ grade ]; // new highest grade
80     } // end for
81
82     return highGrade; // return highest grade
83  } // end function getMaximum
84
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 4 of 8.)

```
85  // determine average grade for test
86  double GradeBook::getAverage()
87  {
88     int total = 0; // initialize total
89
90     // sum grades in array
91     for ( int grade = 0; grade < students; ++grade )
92        total += grades[ grade ];
93
94     // return average of grades
95     return static_cast< double >( total ) / students;
96  } // end function getAverage
97
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 5 of 8.)

```
98  // output bar chart displaying grade distribution
99  void GradeBook::outputBarChart()
100 {
101    cout << "\nGrade distribution:" << endl;
102
103    // stores frequency of grades in each range of 10 grades
104    const int frequencySize = 11;
105    int frequency[ frequencySize ] = {}; // initialize elements to 0
106
107    // for each grade, increment the appropriate frequency
108    for ( int grade = 0; grade < students; ++grade )
109       ++frequency[ grades[ grade ] / students ];
110
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 6 of 8.)

```
111     // for each grade frequency, print bar in chart
112     for ( int count = 0; count < frequencySize; ++count )
113     {
114        // output bar labels ("0-9:", ..., "90-99:", "100:" )
115        if ( count == 0 )
116           cout << "  0-9: ";
117        else if ( count == 10 )
118           cout << "  100: ";
119        else
120           cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
121
122        // print bar of asterisks
123        for ( int stars = 0; stars < frequency[ count ]; ++stars )
124           cout << '*';
125
126        cout << endl; // start a new line of output
127     } // end outer for
128  } // end function outputBarChart
129
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 7 of 8.)

```
130 // output the contents of the grades array
131 void GradeBook::outputGrades()
132 {
133    cout << "\nThe grades are:\n\n";
134
135    // output each student's grade
136    for ( int student = 0; student < students; ++student )
137       cout << "Student " << setw( 2 ) << student + 1 << ": " << setw( 3 )
138          << grades[ student ] << endl;
139 } // end function outputGrades
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 8 of 8.)

7.6 Case Study: Class GradeBook Using an Array to Store Grades (cont.)

- Note that the size of the array is specified as a `public static const` data member.
  - `public` so that it's accessible to the clients of the class.
  - `const` so that this data member is constant.
  - `static` so that the data member is shared by all objects of the class
- There are variables for which each object of a class does not have a separate copy.
- That is the case with `static` data members, which are also known as class variables.
- When objects of a class containing `static` data members are created, all the objects share one copy of the class's `static` data members.

7.6 Case Study: Class GradeBook Using an Array to Store Grades (cont.)

- A `static` data member can be accessed within the class definition and the member-function definitions like any other data member.
- A `public static` data member can also be accessed outside of the class, even when no objects of the class exist, using the class name followed by the binary scope resolution operator (`::`) and the name of the data member.

```
1  // Fig. 7.17: fig07_17.cpp
2  // Creates GradeBook object using an array of grades.
3  #include "GradeBook.h" // GradeBook class definition
4
5  // function main begins program execution
6  int main()
7  {
8     // array of student grades
9     int gradesArray[ GradeBook::students ] =
10       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12    GradeBook myGradeBook(
13       "CS101 Introduction to C++ Programming", gradesArray );
14    myGradeBook.displayMessage();
15    myGradeBook.processGrades();
16 } // end main
```

Fig. 7.17 | Creates a GradeBook object using an array of grades, then invokes member function processGrades to analyze them. (Part 1 of 3.)

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87
```

Fig. 7.17 | Creates a GradeBook object using an array of grades, then invokes member function processGrades to analyze them. (Part 2 of 3.)

```
Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.17 | Creates a GradeBook object using an array of grades, then invokes member function processGrades to analyze them. (Part 3 of 3.)

7.7 Searching Arrays with Linear Search

- Often it may be necessary to determine whether an array contains a value that matches a certain key value.
  - Called searching.
- The linear search compares each element of an array with a search key (line 36).
  - Because the array is not in any particular order, it's just as likely that the value will be found in the first element as the last.
  - On average, therefore, the program must compare the search key with half the elements of the array.
- To determine that a value is not in the array, the program must compare the search key to every element of the array.

```
1  // Fig. 7.18: fig07_18.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using namespace std;
5
6  int linearSearch( const int [], int, int ); // prototype
7
```

Fig. 7.18 | Linear search of an array. (Part 1 of 3.)

```
 8  int main()
 9  {
10     const int arraySize = 100; // size of array a
11     int a[ arraySize ]; // create array a
12     int searchKey; // value to locate in array a
13
14     for ( int i = 0; i < arraySize; ++i )
15        a[ i ] = 2 * i; // create some data
16
17     cout << "Enter integer search key: ";
18     cin >> searchKey;
19
20     // attempt to locate searchKey in array a
21     int element = linearSearch( a, searchKey, arraySize );
22
23     // display results
24     if ( element != -1 )
25        cout << "Found value in element " << element << endl;
26     else
27        cout << "Value not found" << endl;
28  } // end main
29
```

Fig. 7.18 | Linear search of an array. (Part 2 of 3.)

```
30   // compare key to every element of array until location is
31   // found or until end of array is reached; return subscript of
32   // element if key is found or -1 if key not found
33   int linearSearch( const int array[], int key, int sizeOfArray )
34   {
35      for ( int j = 0; j < sizeOfArray; ++j )
36         if ( array[ j ] == key ) // if found,
37            return j; // return location of key
38
39      return -1; // key not found
40   } // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

Fig. 7.18 | Linear search of an array. (Part 3 of 3.)

7.8 Sorting Arrays with Insertion Sort

- Sorting data
  - placing the data into some particular order such as ascending or descending
  - an intriguing prob-lem that has attracted some of the most intense research efforts in the field of computer science.

7.8 Sorting Arrays with Insertion Sort (cont.)

- Insertion sort—a simple, but inefficient, sorting algorithm.
- The first iteration of this algorithm takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the program *inserts the second element in front of the first element).*
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the i^{th} *iteration of this algorithm, the first i elements in the original array will be sorted.*

```
1  // Fig. 7.19: fig07_19.cpp
2  // This program sorts an array's values into ascending order.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int arraySize = 10; // size of array a
10    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
11    int insert; // temporary variable to hold element to insert
12
13    cout << "Unsorted array:\n";
14
15    // output original array
16    for ( int i = 0; i < arraySize; ++i )
17       cout << setw( 4 ) << data[ i ];
18
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 1 of 3.)

```
19      // insertion sort
20      // loop over the elements of the array
21      for ( int next = 1; next < arraySize; ++next )
22      {
23         insert = data[ next ]; // store the value in the current element
24
25         int moveItem = next; // initialize location to place element
26
27         // search for the location in which to put the current element
28         while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
29         {
30            // shift element one slot to the right
31            data[ moveItem ] = data[ moveItem - 1 ];
32            moveItem--;
33         } // end while
34
35         data[ moveItem ] = insert; // place inserted element into the array
36      } // end for
37
38      cout << "\nSorted array:\n";
39
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 2 of 3.)

```
40      // output sorted array
41      for ( int i = 0; i < arraySize; ++i )
42         cout << setw( 4 ) << data[ i ];
43
44      cout << endl;
45   } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 3 of 3.)

7.9 Multidimensional Arrays

- Arrays with two dimensions (i.e., subscripts) often represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we must specify two subscripts.
  - By convention, the first identifies the element's row and the second identifies the element's column.
- Often called two-dimensional arrays or 2-D arrays.
- Arrays with two or more dimensions are known as multidimensional arrays.
- Figure 7.20 illustrates a two-dimensional array, a.
  - The array contains three rows and four columns, so it's said to be a 3-by-4 array.
  - In general, an array with *m rows and n columns is called an m-by-n array*.

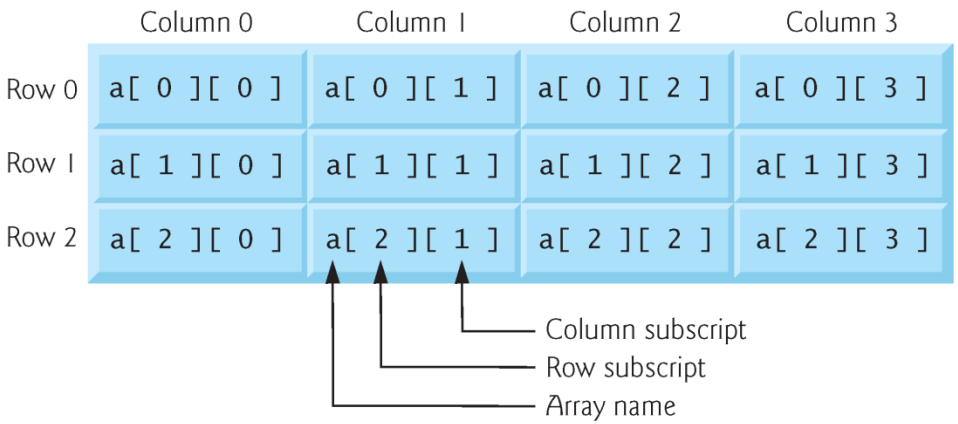

Fig. 7.20 | Two-dimensional array with three rows and four columns.

Common Programming Error 7.7

Referencing a two-dimensional array element `a[x][y]` incorrectly as `a[x, y]` is an error. Actually, `a[x, y]` is treated as `a[y]`, because C++ evaluates the expression `x, y` (containing a comma operator) simply as `y` (the last of the comma-separated expressions).

7.9 Multidimensional Arrays (cont.)

- A multidimensional array can be initialized in its declaration much like a one-dimensional array.
- The values are grouped by row in braces.
- If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.
- Figure 7.21 demonstrates initializing two-dimensional arrays in declarations.

```
1  // Fig. 7.21: fig07_21.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using namespace std;
5
6  void printArray( const int [][ 3 ] ); // prototype
7  const int rows = 2;
8  const int columns = 3;
9
10 int main()
11 {
12    int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13    int array2[ rows ][ columns ] = { 1, 2, 3, 4, 5 };
14    int array3[ rows ][ columns ] = { { 1, 2 }, { 4 } };
15
16    cout << "Values in array1 by row are:" << endl;
17    printArray( array1 );
18
19    cout << "\nValues in array2 by row are:" << endl;
20    printArray( array2 );
21
22    cout << "\nValues in array3 by row are:" << endl;
23    printArray( array3 );
24 } // end main
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 1 of 3.)

```
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ columns ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < rows; ++i )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < columns; ++j )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 2 of 3.)

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 3 of 3.)

7.10 Case Study: Class `GradeBook` Using a Two-Dimensional Array

- In most semesters, students take several exams.
- Professors are likely to want to analyze grades across the entire semester, both for a single student and for the class as a whole.
- Figures 7.22–7.23 contain a version of class `GradeBook` that uses a two-dimensional array `grades` to store the grades of a number of students on multiple exams.
- Each row of the array represents a single student's grades for the entire course, and each column represents all the grades the students earned for one particular exam.

```
1  // Fig. 7.22: GradeBook.h
2  // Definition of class GradeBook that uses a
3  // two-dimensional array to store test grades.
4  // Member functions are defined in GradeBook.cpp
5  #include <string> // program uses C++ Standard Library string class
6  using namespace std;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constants
13    static const int students = 10; // number of students
14    static const int tests = 3; // number of tests
15
16    // constructor initializes course name and array of grades
17    GradeBook( string, const int [][ tests ] );
```

Fig. 7.22 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 1 of 2.)

```
18
19     void setCourseName( string ); // function to set the course name
20     string getCourseName(); // function to retrieve the course name
21     void displayMessage(); // display a welcome message
22     void processGrades(); // perform various operations on the grade data
23     int getMinimum(); // find the minimum grade in the grade book
24     int getMaximum(); // find the maximum grade in the grade book
25     double getAverage( const int [], const int ); // get student's average
26     void outputBarChart(); // output bar chart of grade distribution
27     void outputGrades(); // output the contents of the grades array
28  private:
29     string courseName; // course name for this grade book
30     int grades[ students ][ tests ]; // two-dimensional array of grades
31  }; // end class GradeBook
```

Fig. 7.22 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 2 of 2.)

```
1  // Fig. 7.23: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a two-dimensional array to store grades.
4  #include <iostream>
5  #include <iomanip> // parameterized stream manipulators
6  using namespace std;
7
8  // include definition of class GradeBook from GradeBook.h
9  #include "GradeBook.h"
10
11 // two-argument constructor initializes courseName and grades array
12 GradeBook::GradeBook( string name, const int gradesArray[][ tests ] )
13 {
14    setCourseName( name ); // initialize courseName
15
16    // copy grades from gradeArray to grades
17    for ( int student = 0; student < students; ++student )
18
19       for ( int test = 0; test < tests; ++test )
20          grades[ student ][ test ] = gradesArray[ student ][ test ];
21 } // end two-argument GradeBook constructor
22
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 1 of 10.)

```
23  // function to set the course name
24  void GradeBook::setCourseName( string name )
25  {
26     courseName = name; // store the course name
27  } // end function setCourseName
28
29  // function to retrieve the course name
30  string GradeBook::getCourseName()
31  {
32     return courseName;
33  } // end function getCourseName
34
35  // display a welcome message to the GradeBook user
36  void GradeBook::displayMessage()
37  {
38     // this statement calls getCourseName to get the
39     // name of the course this GradeBook represents
40     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
41        << endl;
42  } // end function displayMessage
43
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 2 of 10.)

```
44  // perform various operations on the data
45  void GradeBook::processGrades()
46  {
47     outputGrades(); // output grades array
48
49     // call functions getMinimum and getMaximum
50     cout << "\nLowest grade in the grade book is " << getMinimum()
51        << "\nHighest grade in the grade book is " << getMaximum() << endl;
52
53     outputBarChart(); // display distribution chart of grades on all tests
54  } // end function processGrades
55
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 3 of 10.)

```
56  // find minimum grade in the entire gradebook
57  int GradeBook::getMinimum()
58  {
59     int lowGrade = 100; // assume lowest grade is 100
60
61     // loop through rows of grades array
62     for ( int student = 0; student < students; ++student )
63     {
64        // loop through columns of current row
65        for ( int test = 0; test < tests; ++test )
66        {
67           // if current grade less than lowGrade, assign it to lowGrade
68           if ( grades[ student ][ test ] < lowGrade )
69              lowGrade = grades[ student ][ test ]; // new lowest grade
70        } // end inner for
71     } // end outer for
72
73     return lowGrade; // return lowest grade
74  } // end function getMinimum
75
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 4 of 10.)

```
76  // find maximum grade in the entire gradebook
77  int GradeBook::getMaximum()
78  {
79     int highGrade = 0; // assume highest grade is 0
80
81     // loop through rows of grades array
82     for ( int student = 0; student < students; ++student )
83     {
84        // loop through columns of current row
85        for ( int test = 0; test < tests; ++test )
86        {
87           // if current grade greater than highGrade, assign to highGrade
88           if ( grades[ student ][ test ] > highGrade )
89              highGrade = grades[ student ][ test ]; // new highest grade
90        } // end inner for
91     } // end outer for
92
93     return highGrade; // return highest grade
94  } // end function getMaximum
95
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 5 of 10.)

```
96  // determine average grade for particular set of grades
97  double GradeBook::getAverage( const int setOfGrades[], const int grades )
98  {
99     int total = 0; // initialize total
100
101    // sum grades in array
102    for ( int grade = 0; grade < grades; ++grade )
103       total += setOfGrades[ grade ];
104
105    // return average of grades
106    return static_cast< double >( total ) / grades;
107 } // end function getAverage
108
109 // output bar chart displaying grade distribution
110 void GradeBook::outputBarChart()
111 {
112    cout << "\nOverall grade distribution:" << endl;
113
114    // stores frequency of grades in each range of 10 grades
115    const int frequencySize = 11;
116    int frequency[ frequencySize ] = {}; // initialize elements to 0
117
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 6 of 10.)

```
118    // for each grade, increment the appropriate frequency
119    for ( int student = 0; student < students; ++student )
120
121       for ( int test = 0; test < tests; ++test )
122          ++frequency[ grades[ student ][ test ] / 10 ];
123
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 7 of 10.)

```
124     // for each grade frequency, print bar in chart
125     for ( int count = 0; count < frequencySize; ++count )
126     {
127        // output bar label ("0-9:", ..., "90-99:", "100:" )
128        if ( count == 0 )
129           cout << "  0-9: ";
130        else if ( count == 10 )
131           cout << "  100: ";
132        else
133           cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
134
135        // print bar of asterisks
136        for ( int stars = 0; stars < frequency[ count ]; ++stars )
137           cout << '*';
138
139        cout << endl; // start a new line of output
140     } // end outer for
141  } // end function outputBarChart
142
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 8 of 10.)

```
143 // output the contents of the grades array
144 void GradeBook::outputGrades()
145 {
146    cout << "\nThe grades are:\n\n";
147    cout << "            "; // align column heads
148
149    // create a column heading for each of the tests
150    for ( int test = 0; test < tests; ++test )
151       cout << "Test " << test + 1 << "  ";
152
153    cout << "Average" << endl; // student average column heading
154
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 9 of 10.)

```
155     // create rows/columns of text representing array grades
156     for ( int student = 0; student < students; ++student )
157     {
158        cout << "Student " << setw( 2 ) << student + 1;
159
160        // output student's grades
161        for ( int test = 0; test < tests; ++test )
162           cout << setw( 8 ) << grades[ student ][ test ];
163
164        // call member function getAverage to calculate student's average;
165        // pass row of grades and the value of tests as the arguments
166        double average = getAverage( grades[ student ], tests );
167        cout << setw( 9 ) << setprecision( 2 ) << fixed << average << endl;
168     } // end outer for
169  } // end function outputGrades
```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 10 of 10.)

```
1  // Fig. 7.24: fig07_24.cpp
2  // Creates GradeBook object using a two-dimensional array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
```

Fig. 7.24 | Creates a GradeBook object using a two-dimensional array of grades, then invokes member function processGrades to analyze them. (Part 1 of 4.)

```
 6  // function main begins program execution
 7  int main()
 8  {
 9     // two-dimensional array of student grades
10     int gradesArray[ GradeBook::students ][ GradeBook::tests ] =
11        { { 87, 96, 70 },
12          { 68, 87, 90 },
13          { 94, 100, 90 },
14          { 100, 81, 82 },
15          { 83, 65, 85 },
16          { 78, 87, 65 },
17          { 85, 75, 83 },
18          { 91, 94, 100 },
19          { 76, 72, 84 },
20          { 87, 93, 73 } };
21
22     GradeBook myGradeBook(
23        "CS101 Introduction to C++ Programming", gradesArray );
24     myGradeBook.displayMessage();
25     myGradeBook.processGrades();
26  } // end main
```

Fig. 7.24 | Creates a GradeBook object using a two-dimensional array of grades, then invokes member function processGrades to analyze them. (Part 2 of 4.)

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100
```

Fig. 7.24 | Creates a GradeBook object using a two-dimensional array of grades, then invokes member function processGrades to analyze them. (Part 3 of 4.)

```
Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.24 | Creates a GradeBook object using a two-dimensional array of grades, then invokes member function processGrades to analyze them. (Part 4 of 4.)

7.11 Introduction to C++ Standard Library Class Template `vector`

- C++ Standard Library class template `vector` represents a more robust type of array featuring many additional capabilities.
- C-style pointer-based arrays have great potential for errors and are not flexible
  - A program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array.
  - Two arrays cannot be meaningfully compared with equality operators or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the size of the array must be passed as an additional argument.
  - One array cannot be assigned to another with the assignment operator.
- Class template `vector` allows you to create a more powerful and less error-prone alternative to arrays.

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- The program of Fig. 7.25 demonstrates capabilities provided by class template `vector` that are not available for C-style pointer-based arrays.
- Standard class template `vector` is defined in header `<vector>` and belongs to namespace `std`.

```
1  // Fig. 7.25: fig07_25.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>
6  using namespace std;
7
8  void outputVector( const vector< int > & ); // display the vector
9  void inputVector( vector< int > & ); // input values into the vector
10
11 int main()
12 {
13    vector< int > integers1( 7 ); // 7-element vector< int >
14    vector< int > integers2( 10 ); // 10-element vector< int >
15
16    // print integers1 size and contents
17    cout << "Size of vector integers1 is " << integers1.size()
18       << "\nvector after initialization:" << endl;
19    outputVector( integers1 );
20
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 1 of 10.)

```
21      // print integers2 size and contents
22      cout << "\nSize of vector integers2 is " << integers2.size()
23         << "\nvector after initialization:" << endl;
24      outputVector( integers2 );
25
26      // input and print integers1 and integers2
27      cout << "\nEnter 17 integers:" << endl;
28      inputVector( integers1 );
29      inputVector( integers2 );
30
31      cout << "\nAfter input, the vectors contain:\n"
32         << "integers1:" << endl;
33      outputVector( integers1 );
34      cout << "integers2:" << endl;
35      outputVector( integers2 );
36
37      // use inequality (!=) operator with vector objects
38      cout << "\nEvaluating: integers1 != integers2" << endl;
39
40      if ( integers1 != integers2 )
41         cout << "integers1 and integers2 are not equal" << endl;
42
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 2 of 10.)

```
43      // create vector integers3 using integers1 as an
44      // initializer; print size and contents
45      vector< int > integers3( integers1 ); // copy constructor
46
47      cout << "\nSize of vector integers3 is " << integers3.size()
48         << "\nvector after initialization:" << endl;
49      outputVector( integers3 );
50
51      // use overloaded assignment (=) operator
52      cout << "\nAssigning integers2 to integers1:" << endl;
53      integers1 = integers2; // assign integers2 to integers1
54
55      cout << "integers1:" << endl;
56      outputVector( integers1 );
57      cout << "integers2:" << endl;
58      outputVector( integers2 );
59
60      // use equality (==) operator with vector objects
61      cout << "\nEvaluating: integers1 == integers2" << endl;
62
63      if ( integers1 == integers2 )
64         cout << "integers1 and integers2 are equal" << endl;
65
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 3 of 10.)

```
66      // use square brackets to create rvalue
67      cout << "\nintegers1[5] is " << integers1[ 5 ];
68
69      // use square brackets to create lvalue
70      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
71      integers1[ 5 ] = 1000;
72      cout << "integers1:" << endl;
73      outputVector( integers1 );
74
75      // attempt to use out-of-range subscript
76      try
77      {
78         cout << "\nAttempt to display integers1.at( 15 )" << endl;
79         cout << integers1.at( 15 ) << endl; // ERROR: out of range
80      } // end try
81      catch ( out_of_range &ex )
82      {
83         cout << "An exception occurred: " << ex.what() << endl;
84      } // end catch
85   } // end main
86
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 5 of 10.)

```
87  // output vector contents
88  void outputVector( const vector< int > &array )
89  {
90     size_t i; // declare control variable
91
92     for ( i = 0; i < array.size(); ++i )
93     {
94        cout << setw( 12 ) << array[ i ];
95
96        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97           cout << endl;
98     } // end for
99
100    if ( i % 4 != 0 )
101       cout << endl;
102 } // end function outputVector
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 6 of 10.)

```
103
104 // input vector contents
105 void inputVector( vector< int > &array )
106 {
107    for ( size_t i = 0; i < array.size(); ++i )
108       cin >> array[ i ];
109 } // end function inputVector
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 7 of 10.)

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Fig. 7.25 | Demonstrating C++ Standard Library class template

```
Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 9 of 10.)

```
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to display integers1.at( 15 )
An exception occurred: invalid vector<T> subscript
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 10 of 10.)

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- By default, all the elements of a `vector` object are set to 0.
- `vector`s can be defined to store any data type.
- `vector` member function `size` obtain the number of elements in the `vector`.
- You can use square brackets, `[]`, to access the elements in a `vector`.
- `vector` objects can be compared with one another using the equality operators.
- You can create a new `vector` object that is initialized with the contents of an existing `vector` by using its copy constructor.

7.11 Introduction to C++ Standard Library Class Template `vector` (cont.)

- You can use the assignment (=) operator with `vector` objects.
- As with C-style pointer-based arrays, C++ does not perform any bounds checking when `vector` elements are accessed with square brackets.
- Standard class template `vector` provides bounds checking in its member function `at`, which "throws an exception" (see Chapter 16, Exception Handling) if its argument is an invalid subscript.