Class Templates `array` and `vector`; Catching Exceptions

Chapter 7 of C++ How to Program, 10/e

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

OBJECTIVES

In this chapter you'll:

- Use C++ Standard Library class template `array`—a fixed-size collection of related data items.
- Declare `arrays`, initialize `arrays` and refer to the elements of `arrays`.
- Use `arrays` to store, sort and search lists and tables of values.
- Use the range-based `for` statement.
- Pass `arrays` to functions.
- Use C++ Standard Library function `sort` to arrange `array` elements in ascending order.
- Use C++ Standard Library function `binary_search` to locate an element in a sorted `array`.
- Declare and manipulate multidimensional `arrays`.
- Use one- and two-dimensional `arrays` to build a real-world `GradeBook` class.
- Use C++ Standard Library class template `vector`—a variable-size collection of related data items.

7.1 Introduction

7.2 `arrays`

7.3 Declaring `arrays`

7.4 Examples Using `arrays`

- 7.4.1 Declaring an `array` and Using a Loop to Initialize the `array`'s Elements
- 7.4.2 Initializing an `array` in a Declaration with an Initializer List
- 7.4.3 Specifying an `array`'s Size with a Constant Variable and Setting `array` Elements with Calculations
- 7.4.4 Summing the Elements of an `array`
- 7.4.5 Using a Bar Chart to Display `array` Data Graphically
- 7.4.6 Using the Elements of an `array` as Counters
- 7.4.7 Using `arrays` to Summarize Survey Results
- 7.4.8 Static Local `arrays` and Automatic Local `arrays`

7.5 Range-Based `for` Statement

7.6 Case Study: Class `GradeBook` Using an `array` to Store Grades

7.7 Sorting and Searching `arrays`

- 7.7.1 Sorting
- 7.7.2 Searching
- 7.7.3 Demonstrating Functions `sort` and `binary_search`

7.8 Multidimensional `arrays`
7.9 Case Study: Class `GradeBook` Using a Two-Dimensional `array`
7.10 Introduction to C++ Standard Library Class Template `vector`
7.11 Wrap-Up

7.1 Introduction

- This chapter introduces the topic of data structures—*collections* of related data items.
- We discuss `arrays` which are *fixed-size* collections consisting of data items of the *same* type, and `vectors` which are collections (also of data items of the *same* type) that can grow and shrink *dynamically* at execution time.
- Both `array` and `vector` are C++ standard library class templates.
- We present examples that demonstrate several common array manipulations and introduce exception handling.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.2 arrays

- An array is a *contiguous* group of memory locations that all have the same type.
- To refer to a particular location or element in the array, specify the name of the array and the position number of the particular element.
- Figure 7.1 shows an integer array called c.
- 12 elements.
- The position number is more formally called a subscript or index (this number specifies the number of elements from the beginning of the array).
- The first element in every array has subscript 0 (zero) and is sometimes called the zeroth element.
- The highest subscript in array c is 11, which is 1 less than the number of elements in the array (12).
- A subscript must be an integer or integer expression (using any integral type).

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

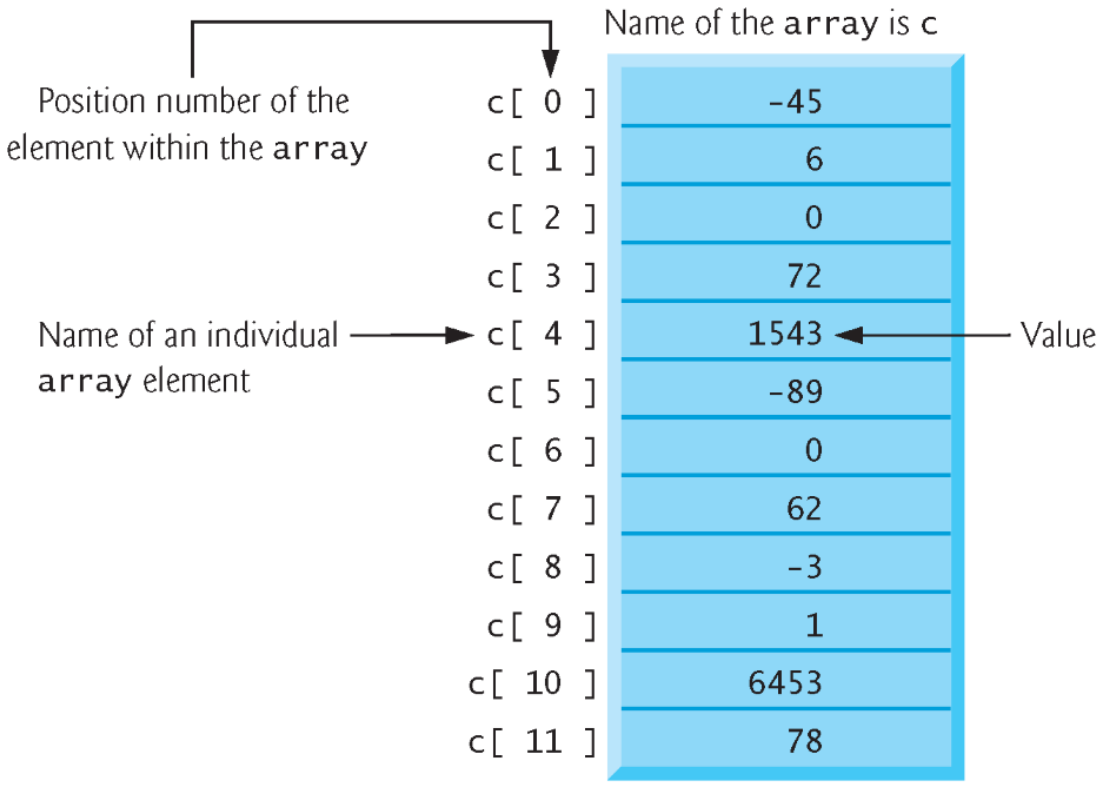

Fig. 7.1 | array of 12 elements.

Common Programming Error 7.1

Note the difference between the "seventh element of the `array` *" and "*`array` *element 7." Subscripts begin at 0, so the "seventh element of the* `array` *" has the subscript 6, while "*`array` *element 7" has the subscript 7 and is actually the* eighth *element of the* `array`*. This distinction is a frequent source of* ***off-by-one errors****. To avoid such errors, we refer to specific* `array` *elements explicitly by their* `array` *name and subscript number (e.g.,* `c[6]` *or* `c[7]`*).*

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`()` `[]` `++` `--` `static_cast<`*type*`>(`*operand*`)`	left to right	postfix
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.2 | Precedence and associativity of the operators introduced to this point.

7.3 Declaring arrays

- `arrays` occupy space in memory.
- To specify the type of the elements and the number of elements required by an array use a declaration of the form:
  - `array`< *type, arraySize* > *arrayName*`;`
- The notation <*type, arraySize*> indicates that `array` is a class template.
- The compiler reserves the appropriate amount of memory based on the *type* of the elements and the *arraySize*.
- `arrays` can be declared to contain values of most data types.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.4 Examples Using arrays

- The following examples demonstrate how to declare arrays, how to initialize arrays and how to perform common array manipulations.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.4.1 Declaring an `array` and Using a Loop to Initialize the `array`'s Elements

- The program in Fig. 7.3 declares five-element integer array `n` (line 10).
- `size_t` represents an unsigned integral type.
- This type is recommended for any variable that represents an `array`'s size or an `array`'s subscripts. Type `size_t` is defined in the `std` namespace and is in header `<cstddef>`, which is included by various other headers.
- If you attempt to compile a program that uses type `size_t` and receive errors indicating that it's not defined, simply include `<cstddef>` in your program.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
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  #include <array>
6  using namespace std;
7
8  int main() {
9     array<int, 5> n; // n is an array of 5 int values
10
11    // initialize elements of array n to 0
12    for (size_t i{0}; i < n.size(); ++i) {
13       n[i] = 0; // set element at location i to 0
14    }
15
16    cout << "Element" << setw(10) << "Value" << endl;
17
18    // output each array element's value
19    for (size_t j{0}; j < n.size(); ++j) {
20       cout << setw(7) << j << setw(10) << n[j] << endl;
21    }
22 }
```

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

```
Element     Value
      0         0
      1         0
      2         0
      3         0
      4         0
```

Fig. 7.3 | Initializing an `array`'s elements to zeros and printing the `array`. (Part 2 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 five values (line 11) and prints the array in tabular format (lines 13–17).
- If there are *fewer* initializers than elements in the `array`, the remaining `array` elements are initialized to zero.
- If there are more, a compilation error occurs.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

```
Element     Value
      0        32
      1        27
      2        64
      3        18
      4        95
```

Fig. 7.4 | Initializing an `array` in a declaration.

7.4.3 Specifying an `array`'s Size with a Constant Variable and Setting `array` Elements with Calculations

- Figure 7.5 sets the elements of a 5-element array `values` to the even integers `2`, `4`, `6`, `8` and `10` and prints the array in tabular format.
- Line 10 uses the `const` qualifier to declare a constant variable `arraySize` with the value `5`.
  - A constant variable that's used to specify `array`'s size *must* be initialized with a constant expression when it's declared and *cannot* be modified thereafter.
- Constant variables are also called named constants or read-only variables.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 10.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main() {
9     // constant variable can be used to specify array size
10    const size_t arraySize{5}; // must initialize in declaration
11
12    array<int, arraySize> values; // array values has 5 elements
13
14    for (size_t i{0}; i < values.size(); ++i) { // set the values
15       values[i] = 2 + 2 * i;
16    }
17
18    cout << "Element" << setw(10) << "Value" << endl;
19
20    // output contents of array s in tabular format
21    for (size_t j{0}; j < values.size(); ++j) {
22       cout << setw(7) << j << setw(10) << values[j] << endl;
23    }
24 }
```

Fig. 7.5 | Set `array` `s` to the even integers from 2 to 10. (Part 1 of 2.)

```
Element     Value
      0         2
      1         4
      2         6
      3         8
      4        10
```

Fig. 7.5 | Set `array s` to the even integers from 2 to 10. (Part 2 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 a separate statement from its declaration is a compilation error.

Good Programming Practice 7.1

Defining the size of an `array` *as a constant variable instead of a literal constant makes programs clearer and easier to update. This technique eliminates so-called* ***magic numbers****—numeric values that are not explained. Using a constant variable allows you to provide a name for a literal constant and can help explain the purpose of the value in the program.*

7.4.4 Summing the Elements of an `array`

- Often, the elements of an array represent a series of values to be used in a calculation.
- Fig. 7.6 sums the values contained in the four-element integer `array a`.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.6: fig07_06.cpp
2  // Compute the sum of the elements of an array.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  int main() {
8     const size_t arraySize{4}; // specifies size of array
9     array<int, arraySize> a{10, 20, 30, 40};
10    int total{0};
11
12    // sum contents of array a
13    for (size_t i{0}; i < a.size(); ++i) {
14       total += a[i];
15    }
16
17    cout << "Total of array elements: " << total << endl;
18 }
```

```
Total of array elements: 100
```

Fig. 7.6 | Compute the sum of the elements of an `array`.

7.4.5 Using Bar Charts to Display `array` Data Graphically

- Many programs present data to users graphically.
- One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).
- Fig. 7.7 stores data in an `array` of 11 elements, each corresponding to a grade category.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.7: fig07_07.cpp
2  // Bar chart printing program.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main() {
9     const size_t arraySize{11};
10    array<unsigned int, arraySize> n{0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
11
12    cout << "Grade distribution:" << endl;
13
```

Fig. 7.7 | Bar chart printing program. (Part 1 of 3.)

```
14      // for each element of array n, output a bar of the chart
15      for (size_t i{0}; i < n.size(); ++i) {
16         // output bar labels ("0-9:", ..., "90-99:", "100:")
17         if (0 == i) {
18            cout << "  0-9: ";
19         }
20         else if (10 == i) {
21            cout << "  100: ";
22         }
23         else {
24            cout << i * 10 << "-" << (i * 10) + 9 << ": ";
25         }
26
27         // print bar of asterisks
28         for (unsigned int stars{0}; stars < n[i]; ++stars) {
29            cout << '*';
30         }
31
32         cout << endl; // start a new line of output
33      }
34   }
```

Fig. 7.7 | Bar chart printing program. (Part 2 of 3.)

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

Fig. 7.7 | Bar chart printing program. (Part 3 of 3.)

7.4.6 Using the Elements of an `array` as Counters

- Sometimes, programs use counter variables to summarize data, such as the results of a survey.
- In Fig. 6.9, we used separate counters in our die-rolling program to track the number of occurrences of each side of a die as the program rolled the die 60,000,000 times.
- An `array` version of this program is shown in Fig. 7.8.
- This version also uses the new C++11 random-number generation capabilities that were introduced in Section 6.9.
- *The single statement in line 21 of this program replaces the* `switch` *statement in Fig. 6.9.*

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.8: fig07_08.cpp
2  // Die-rolling program using an array instead of switch.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  #include <random>
7  #include <ctime>
8  using namespace std;
9
10 int main() {
11    // use the default random-number generation engine to
12    // produce uniformly distributed pseudorandom int values from 1 to 6
13    default_random_engine engine(static_cast<unsigned int>(time(0)));
14    uniform_int_distribution<unsigned int> randomInt(1, 6);
15
16    const size_t arraySize{7}; // ignore element zero
17    array<unsigned int, arraySize> frequency{}; // initialize to 0s
18
19    // roll die 60,000,000 times; use die value as frequency index
20    for (unsigned int roll{1}; roll <= 60'000'000; ++roll) {
21       ++frequency[randomInt(engine)];
22    }
23
```

Fig. 7.8 | Die-rolling program using an `array` instead of `switch`. (Part 1 of 2.)

```
24      cout << "Face" << setw(13) << "Frequency" << endl;
25
26      // output each array element's value
27      for (size_t face{1}; face < frequency.size(); ++face) {
28         cout << setw(4) << face << setw(13) << frequency[face] << endl;
29      }
30   }
```

```
Face     Frequency
   1       9997901
   2       9999110
   3      10001172
   4      10003619
   5       9997606
   6      10000592
```

Fig. 7.8 | Die-rolling program using an `array` instead of `switch`. (Part 2 of 2.)

7.4.7 Using arrays to Summarize Survey Results

- Fig. 7.9 uses arrays to summarize the results of data collected in a survey.
- Consider the following problem statement:
  - Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.
- *C++ provides no automatic* `array` *bounds checking to prevent you from referring to an element that does not exist.*
- Thus, an executing program can "walk off" either end of an `array` without warning.
- Class templates `array` and `vector` each have an `at` function that performs bounds checking for you.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.9: fig07_09.cpp
2  // Poll analysis program.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main() {
9     // define array sizes
10    const size_t responseSize{20}; // size of array responses
11    const size_t frequencySize{6}; // size of array frequency
12
13    // place survey responses in array responses
14    const array<unsigned int, responseSize> responses{
15       1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
16
17    // initialize frequency counters to 0
18    array<unsigned int, frequencySize> frequency{};
19
20    // for each answer, select responses element and use that value
21    // as frequency subscript to determine element to increment
22    for (size_t answer{0}; answer < responses.size(); ++answer) {
23       ++frequency[responses[answer]];
24    }
```

Fig. 7.9 | Poll analysis program. (Part 1 of 2.)

```
25
26      cout << "Rating" << setw(12) << "Frequency" << endl;
27
28      // output each array element's value
29      for (size_t rating{1}; rating < frequency.size(); ++rating) {
30         cout << setw(6) << rating << setw(12) << frequency[rating] << endl;
31      }
32   }
```

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

Fig. 7.9 | Poll analysis program. (Part 2 of 2.)

7.4.7 Using arrays to Summarize Survey Results

- It's important to ensure that every subscript you use to access an `array` element is within the `array`'s bounds—that is, greater than or equal to 0 and less than the number of `array` elements.
- Allowing programs to read from or write to `array` elements outside the bounds of `arrays` are common *security flaws*.
- Reading from out-of-bounds `array` elements can cause a program to crash or even appear to execute correctly while using bad data.
- Writing to an out-of-bounds element (known as a *buffer overflow*) can corrupt a program's data in memory, crash a program and allow attackers to exploit the system and execute their own code.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 7.4

Referring to an element outside the `array` *bounds is an* execution-time *logic error, not a syntax error.*

Error-Prevention Tip 7.1

When looping through an `array`*, the index should never go below 0 and should always be less than the total number of* `array` *elements (one less than the size of the* `array`*). Make sure that the loop-termination condition prevents accessing elements outside this range. In Section 7.5, you'll learn about the range-based* `for` *statement, which can help prevent accessing elements outside an array's (or other container's) bounds.*

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.
- Local variables are sometimes called automatic variables because they're automatically destroyed when the function finishes executing

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

Performance Tip 7.1

We can apply `static` *to a local* `array` *declaration so that it's* 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. 7.10: fig07_10.cpp
2  // static array initialization and automatic array initialization.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  void staticArrayInit(); // function prototype
8  void automaticArrayInit(); // function prototype
9  const size_t arraySize{3};
10
11 int main() {
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 }
21
```

Fig. 7.10 | `static` array initialization and automatic `array` initialization. (Part 1 of 4.)

```
22  // function to demonstrate a static local array
23  void staticArrayInit(void) {
24     // initializes elements to 0 first time function is called
25     static array<int, arraySize> array1; // static local array
26
27     cout << "\nValues on entering staticArrayInit:\n";
28
29     // output contents of array1
30     for (size_t i{0}; i < array1.size(); ++i) {
31        cout << "array1[" << i << "] = " << array1[i] << "   ";
32     }
33
34     cout << "\nValues on exiting staticArrayInit:\n";
35
36     // modify and output contents of array1
37     for (size_t j{0}; j < array1.size(); ++j) {
38        cout << "array1[" << j << "] = " << (array1[j] += 5) << "   ";
39     }
40  }
41
```

Fig. 7.10 | `static` array initialization and automatic `array` initialization. (Part 2 of 4.)

```
42  // function to demonstrate an automatic local array
43  void automaticArrayInit(void) {
44     // initializes elements each time function is called
45     array<int, arraySize> array2{1, 2, 3}; // automatic local array
46
47     cout << "\n\nValues on entering automaticArrayInit:\n";
48
49     // output contents of array2
50     for (size_t i{0}; i < array2.size(); ++i) {
51        cout << "array2[" << i << "] = " << array2[i] << "   ";
52     }
53
54     cout << "\nValues on exiting automaticArrayInit:\n";
55
56     // modify and output contents of array2
57     for (size_t j{0}; j < array2.size(); ++j) {
58        cout << "array2[" << j << "] = " << (array2[j] += 5) << "   ";
59     }
60  }
```

Fig. 7.10 | `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. 7.10 | `static` array initialization and automatic `array` initialization. (Part 4 of 4.)

7.5 Range-Based `for` Statement

- It's common to process *all* the elements of an `array`.
- The C++11 range-based `for` statement allows you to do this *without using a counter*, thus avoiding the possibility of "stepping outside" the `array` and eliminating the need for you to implement your own bounds checking.
- Figure 7.11 uses the range-based `for` to display an `array`'s contents and to multiply each of the `array`'s element values by 2.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 7.2

When processing all elements of an `array`*, if you don't need access to an* `array` *element's subscript, use the range-based* `for` *statement.*

```
1  // Fig. 7.11: fig07_11.cpp
2  // Using range-based for to multiply an array's elements by 2.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  int main() {
8     array<int, 5> items{1, 2, 3, 4, 5};
9
10    // display items before modification
11    cout << "items before modification: ";
12    for (int item : items) {
13       cout << item << " ";
14    }
15
16    // multiply the elements of items by 2
17    for (int& itemRef : items) {
18       itemRef *= 2;
19    }
20
```

Fig. 7.11 | Using range-based `for` to multiply an `array`'s elements by 2. (Part 1 of 2.)

```
21      // display items after modification
22      cout << "\nitems after modification: ";
23      for (int item : items) {
24         cout << item << " ";
25      }
26
27      cout << endl;
28   }
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10
```

Fig. 7.11 | Using range-based `for` to multiply an `array`'s elements by 2. (Part 2 of 2.)

7.5 Range-Based for Statement (cont.)

Using the Range-Based for to Display an array's Contents

- The range-based `for` statement simplifies the code for iterating through an array.
- Line 12 can be read as "for each iteration, assign the next element of `items` to `int` variable `item`, then execute the following statement."
- Lines 12–14 are equivalent to the following counter-controlled iteration:

```
for (int counter = 0; counter < items.size(); ++counter) {
   cout << items[counter] << " ";
}
```

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.5 Range-Based for Statement (cont.)

Using the Range-Based for to Modify an array's Contents

- Lines 17–19 use a range-based `for` statement to multiply each element of `items` by 2.
- In line 17, the range variable declaration indicates that `itemRef` is an `int` *reference* (`&`).
- We use an `int` reference because items contains `int` values and we want to modify each element's value—because `itemRef` is declared as a reference, any change you make to `itemRef` changes the corresponding element value in the `array`.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.5 Range-Based for Statement (cont.)

Using an Element's Subscript

- The range-based `for` statement can be used in place of the counter-controlled `for` statement whenever code looping through an array does not require access to the element's subscript.
- However, if a program must use subscripts for some reason other than simply to loop through an `array` (e.g., to print a subscript number next to each `array` element value, as in the examples early in this chapter), you should use the counter-controlled for statement.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

- Case study on developing a `GradeBook` class that instructors can use to maintain students' grades on an exam and display a grade report that includes the grades, class average, lowest grade, highest grade and a grade distribution bar chart.
- This version stores the grades for one exam in a one-dimensional array.
- In Section 7.9, we present a version that uses a two-dimensional array to store students' grades for several exams.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
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

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

Fig. 7.12 | Output of the `GradeBook` example that stores grades in an `array`. (Part 1 of 2.)

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

Fig. 7.12 | Output of the `GradeBook` example that stores grades in an `array`. (Part 2 of 2.)

```
1  // Fig. 7.13: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  #include <string>
4  #include <array>
5  #include <iostream>
6  #include <iomanip> // parameterized stream manipulators
7
8  // GradeBook class definition
9  class GradeBook {
10 public:
11    // constant number of students who took the test
12    static const size_t students{10}; // note public data
13
14    // constructor initializes courseName and grades array
15    GradeBook(const std::string& name,
16       const std::array<int, students>& gradesArray)
17       : courseName{name}, grades{gradesArray} {
18    }
19
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 1 of 9.)

```
20      // function to set the course name
21      void setCourseName(const std::string& name) {
22         courseName = name; // store the course name
23      }
24
25      // function to retrieve the course name
26      const std::string& getCourseName() const {
27         return courseName;
28      }
29
30      // display a welcome message to the GradeBook user
31      void displayMessage() const {
32         // call getCourseName to get the name of this GradeBook's course
33         std::cout << "Welcome to the grade book for\n" << getCourseName()
34            << "!" << std::endl;
35      }
36
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 2 of 9.)

```
37      // perform various operations on the data (none modify the data)
38      void processGrades() const {
39         outputGrades(); // output grades array
40
41         // call function getAverage to calculate the average grade
42         std::cout << std::setprecision(2) << std::fixed;
43         std::cout << "\nClass average is " << getAverage() << std::endl;
44
45         // call functions getMinimum and getMaximum
46         std::cout << "Lowest grade is " << getMinimum()
47            << "\nHighest grade is " << getMaximum() << std::endl;
48
49         outputBarChart(); // display grade distribution chart
50      }
51
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 3 of 9.)

```
52      // find minimum grade
53      int getMinimum() const {
54         int lowGrade{100}; // assume lowest grade is 100
55
56         // loop through grades array
57         for (int grade : grades) {
58            // if current grade lower than lowGrade, assign it to lowGrade
59            if (grade < lowGrade) {
60               lowGrade = grade; // new lowest grade
61            }
62         }
63
64         return lowGrade; // return lowest grade
65      }
66
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 4 of 9.)

```
67      // find maximum grade
68      int getMaximum() const {
69         int highGrade{0}; // assume highest grade is 0
70
71         // loop through grades array
72         for (int grade : grades) {
73            // if current grade higher than highGrade, assign it to highGrade
74            if (grade > highGrade) {
75               highGrade = grade; // new highest grade
76            }
77         }
78
79         return highGrade; // return highest grade
80      }
81
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 5 of 9.)

```
82      // determine average grade for test
83      double getAverage() const {
84         int total{0}; // initialize total
85
86         // sum grades in array
87         for (int grade : grades) {
88            total += grade;
89         }
90
91         // return average of grades
92         return static_cast<double>(total) / grades.size();
93      }
94
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 6 of 9.)

```
95      // output bar chart displaying grade distribution
96      void outputBarChart() const {
97         std::cout << "\nGrade distribution:" << std::endl;
98
99         // stores frequency of grades in each range of 10 grades
100        const size_t frequencySize{11};
101        std::array<unsigned int, frequencySize> frequency{}; // init to 0s
102
103        // for each grade, increment the appropriate frequency
104        for (int grade : grades) {
105           ++frequency[grade / 10];
106        }
107
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 7 of 9.)

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

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 8 of 9.)

```
130     // output the contents of the grades array
131     void outputGrades() const {
132        std::cout << "\nThe grades are:\n\n";
133
134        // output each student's grade
135        for (size_t student{0}; student < grades.size(); ++student) {
136           std::cout << "Student " << std::setw(2) << student + 1 << ": "
137              << std::setw(3) << grades[student] << std::endl;
138        }
139     }
140  private:
141     std::string courseName; // course name for this grade book
142     std::array<int, students> grades; // array of student grades
143  };
```

Fig. 7.13 | Definition of class GradeBook that uses an array to store test grades. (Part 9 of 9.)

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

- The size of the array is specified as a `public static const` data member `students`.
  - `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's 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.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.14: fig07_14.cpp
2  // Creates GradeBook object using an array of grades.
3  #include <array>
4  #include "GradeBook.h" // GradeBook class definition
5  using namespace std;
6
7  int main() {
8     // array of student grades
9     const array<int, GradeBook::students> grades{
10       87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
11    string courseName{"CS101 Introduction to C++ Programming"};
12
13    GradeBook myGradeBook(courseName, grades);
14    myGradeBook.displayMessage();
15    myGradeBook.processGrades();
16 }
```

Fig. 7.14 | Creates a `GradeBook` object' using an `array` of grades, then invokes member function `processGrades` to analyze them.

7.7 Sorting and Searching arrays

- In this section, we use the built-in C++ Standard Library `sort` function to arrange the elements in an `array` into ascending order and the built-in `binary_search` function to determine whether a value is in the `array`.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.7.1 Sorting

- Sorting data—placing it into ascending or descending order—is one of the most important computing applications.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.7.2 Searching

- Often it may be necessary to determine whether an `array` contains a value that matches a certain key value.
- This is called searching.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.7.2 Searching

- Figure 7.15 begins by creating an unsorted `array` of `strings` and displaying the contents of the `array`.
- Line 21 uses C++ Standard Library function `sort` to sort the elements of the `array` colors into ascending order.
- Lines 25–276 display the contents of the sorted `array`.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.7 Sorting and Searching arrays (cont.)

- Lines 30 and 354 demonstrate use `binary_search` to determine whether a value is in the `array`.
- The sequence of values must be sorted in ascending order first—`binary_search` does *not* verify this for you.
- The function's first two arguments represent the range of elements to search and the third is the *search key*—the value to locate in the `array`.
- The function returns a `bool` indicating whether the value was found.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.15: fig07_15.cpp
2  // Sorting and searching arrays.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  #include <string>
7  #include <algorithm> // contains sort and binary_search
8  using namespace std;
9
10 int main() {
11    const size_t arraySize{7}; // size of array colors
12    array<string, arraySize> colors{"red", "orange", "yellow",
13       "green", "blue", "indigo", "violet"};
14
15    // output original array
16    cout << "Unsorted array:\n";
17    for (string color : colors) {
18       cout << color << " ";
19    }
20
21    sort(colors.begin(), colors.end()); // sort contents of colors
22
```

Fig. 7.15 | Sorting and searching arrays. (Part 1 of 2.)

```
23      // output sorted array
24      cout << "\nSorted array:\n";
25      for (string item : colors) {
26         cout << item << " ";
27      }
28
29      // search for "indigo" in colors
30      bool found{binary_search(colors.begin(), colors.end(), "indigo")};
31      cout << "\n\n\"indigo\" " << (found ? "was" : "was not")
32         << " found in colors" << endl;
33
34      // search for "cyan" in colors
35      found = binary_search(colors.begin(), colors.end(), "cyan");
36      cout << "\"cyan\" " << (found ? "was" : "was not")
37         << " found in colors" << endl;
38   }
```

```
Unsorted array:
red orange yellow green blue indigo violet
Sorted array:
blue green indigo orange red violet yellow

"indigo" was found in colors
"cyan" was not found in colors
```

Fig. 7.15 | Sorting and searching `arrays`. (Part 2 of 2.)

7.8 Multidimensional Arrays

- You can use `arrays` with two dimensions (i.e., subscripts) to 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 7.16 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.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

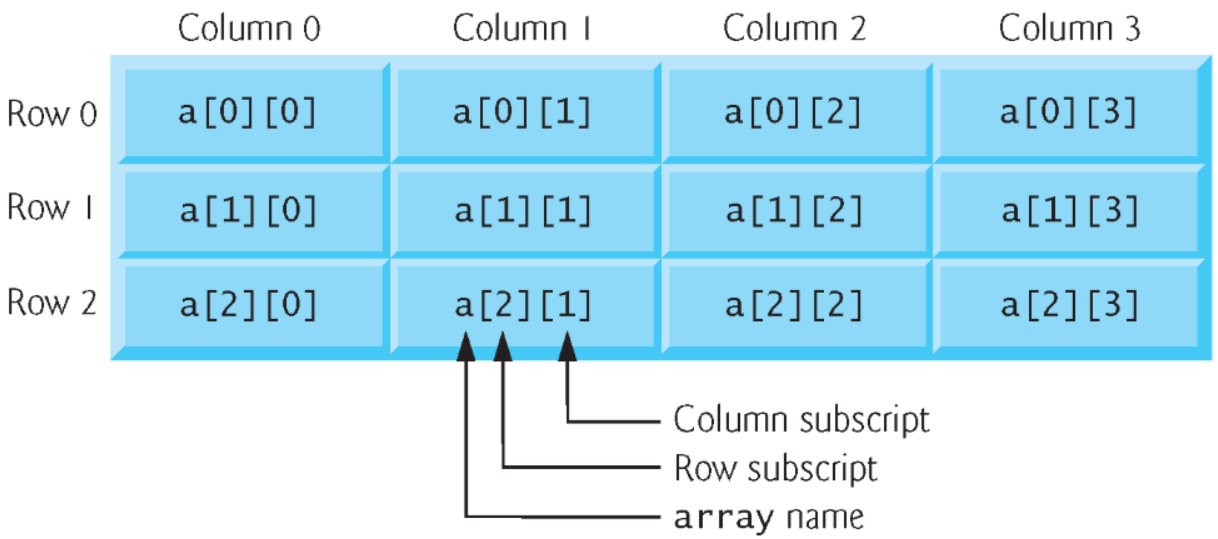

Fig. 7.16 | Two-dimensional `array` with three rows and four columns.

Common Programming Error 7.5

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).*

7.8 Multidimensional arrays (cont.)

- Figure 7.17 demonstrates initializing two-dimensional `arrays` in declarations.
- In each `array`, the type of its elements is specified as
  - `array<int, columns>`
- Each `array` contains as its elements three-element `arrays` of `int` values—the constant `columns` has the value 3.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.17: fig07_17.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  const size_t rows{2};
8  const size_t columns{3};
9  void printArray(const array<array<int, columns>, rows>&);
10
11 int main() {
12    array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};
13    array<array<int, columns>, rows> array2{1, 2, 3, 4, 5};
14
15    cout << "Values in array1 by row are:" << endl;
16    printArray(array1);
17
18    cout << "\nValues in array2 by row are:" << endl;
19    printArray(array2);
20 }
21
```

Fig. 7.17 | Initializing multidimensional arrays. (Part 1 of 2.)

```
22  // output array with two rows and three columns
23  void printArray(const array<array<int, columns>, rows>& a) {
24     // loop through array's rows
25     for (auto const& row : a) {
26        // loop through columns of current row
27        for (auto const& element : row) {
28           cout << element << ' ';
29        }
30
31        cout << endl; // start new line of output
32     }
33  }
```

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

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

Fig. 7.17 | Initializing multidimensional `arrays`. (Part 2 of 2.)

7.8 Multidimensional arrays (cont.)

Nested Range-Based for Statements

- To process the elements of a two-dimensional `array`, we use a nested loop in which the *outer* loop iterates through the *rows* and the *inner* loop iterates through the *columns* of a given row.
- The C++11 auto keyword tells the compiler to infer (determine) a variable's data type based on the variable's initializer value.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.8 Multidimensional arrays (cont.)

Nested Counter-Controlled for Statements

- We could have implemented the nested loop with counter-controlled iteration as follows:

```
for (size_t row = 0; row < a.size(); ++row) {
   for (size_t column = 0; column < a[row].size(); ++column) {
      cout << a[row][column] << ' ';
   }

   cout << endl;
}
```

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

7.9 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.
- Figure 7.18 shows the output that summarizes 10 students grades on three exams.
- We store the grades as a two-dimensional `array` in an object of the next version of class `GradeBook` Figures 7.19–7.20.
- 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.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
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

Overall grade distribution:
  0-9:
10-19:
```

Fig. 7.18 | Output of GradeBook that uses two-dimensional arrays. (Part 1 of 2.)

```
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: **********
90-99: *******
  100: ***
```

Fig. 7.18 | Output of GradeBook that uses two-dimensional `arrays`. (Part 2 of 2.)

```
1  // Fig. 7.19: GradeBook.h
2  // Definition of class GradeBook that uses a
3  // two-dimensional array to store test grades.
4  #include <array>
5  #include <string>
6  #include <iostream>
7  #include <iomanip> // parameterized stream manipulators
8
9  // GradeBook class definition
10 class GradeBook {
11 public:
12    // constants
13    static const size_t students{10}; // number of students
14    static const size_t tests{3}; // number of tests
15
16    // two-argument constructor initializes courseName and grades array
17    GradeBook(const std::string& name,
18       std::array<std::array<int, tests>, students>& gradesArray)
19       : courseName(name), grades(gradesArray) {
20    }
21
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 1 of 10.)

```
22      // function to set the course name
23      void setCourseName(const std::string& name) {
24         courseName = name; // store the course name
25      }
26
27      // function to retrieve the course name
28      const std::string& getCourseName() const {
29         return courseName;
30      }
31
32      // display a welcome message to the GradeBook user
33      void displayMessage() const {
34         // call getCourseName to get this GradeBook's course name
35         std::cout << "Welcome to the grade book for\n" << getCourseName()
36            << "!" << std::endl;
37      }
38
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 2 of 10.)

```
39      // perform various operations on the data
40      void processGrades() const {
41         outputGrades(); // output grades array
42
43         // call functions getMinimum and getMaximum
44         std::cout << "\nLowest grade in the grade book is " << getMinimum()
45            << "\nHighest grade in the grade book is " << getMaximum()
46            << std::endl;
47
48         outputBarChart(); // display grade distribution chart
49      }
50
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 3 of 10.)

```
51      // find minimum grade in the entire gradebook
52      int getMinimum() const {
53         int lowGrade{100}; // assume lowest grade is 100
54
55         // loop through rows of grades array
56         for (auto const& student : grades) {
57            // loop through columns of current row
58            for (auto const& grade : student) {
59               if (grade < lowGrade) { // if grade is lower than lowGrade
60                  lowGrade = grade; // grade is new lowest grade
61               }
62            }
63         }
64
65         return lowGrade; // return lowest grade
66      }
67
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 4 of 10.)

```
68      // find maximum grade in the entire gradebook
69      int getMaximum() const {
70         int highGrade{0}; // assume highest grade is 0
71
72         // loop through rows of grades array
73         for (auto const& student : grades) {
74            // loop through columns of current row
75            for (auto const& grade : student) {
76               if (grade > highGrade) { // if grade is higher than highGrade
77                  highGrade = grade; // grade is new highest grade
78               }
79            }
80         }
81
82         return highGrade; // return highest grade
83      }
84
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 5 of 10.)

```
85      // determine average grade for particular set of grades
86      double getAverage(const std::array<int, tests>& setOfGrades) const {
87         int total{0}; // initialize total
88
89         // sum grades in array
90         for (int grade : setOfGrades) {
91            total += grade;
92         }
93
94         // return average of grades
95         return static_cast<double>(total) / setOfGrades.size();
96      }
97
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 6 of 10.)

```
98      // output bar chart displaying grade distribution
99      void outputBarChart() const {
100        std::cout << "\nOverall grade distribution:" << std::endl;
101
102        // stores frequency of grades in each range of 10 grades
103        const size_t frequencySize{11};
104        std::array<unsigned int, frequencySize> frequency{}; // init to 0s
105
106        // for each grade, increment the appropriate frequency
107        for (auto const& student : grades) {
108           for (auto const& test : student) {
109              ++frequency[test / 10];
110           }
111        }
112
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 7 of 10.)

```
113         // for each grade frequency, print bar in chart
114         for (size_t count{0}; count < frequencySize; ++count) {
115            // output bar label ("0-9:", ..., "90-99:", "100:")
116            if (0 == count) {
117               std::cout << "  0-9: ";
118            }
119            else if (10 == count) {
120               std::cout << "  100: ";
121            }
122            else {
123               std::cout << count * 10 << "-" << (count * 10) + 9 << ": ";
124            }
125
126            // print bar of asterisks
127            for (unsigned int stars{0}; stars < frequency[count]; ++stars)
128               std::cout << '*';
129
130            std::cout << std::endl; // start a new line of output
131         }
132      }
133
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 8 of 10.)

```
134     // output the contents of the grades array
135     void outputGrades() const {
136        std::cout << "\nThe grades are:\n\n";
137        std::cout << "            "; // align column heads
138
139        // create a column heading for each of the tests
140        for (size_t test{0}; test < tests; ++test) {
141           std::cout << "Test " << test + 1 << "  ";
142        }
143
144        std::cout << "Average" << std::endl;
145
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 9 of 10.)

```
146         // create rows/columns of text representing array grades
147         for (size_t student{0}; student < grades.size(); ++student) {
148            std::cout << "Student " << std::setw(2) << student + 1;
149
150            // output student's grades
151            for (size_t test{0}; test < grades[student].size(); ++test) {
152               std::cout << std::setw(8) << grades[student][test];
153            }
154
155            // call member function getAverage to calculate student's
156            // average; pass one row of grades as the argument
157            double average{getAverage(grades[student])};
158            std::cout << std::setw(9) << std::setprecision(2) << std::fixed
159               << average << std::endl;
160         }
161      }
162   private:
163      std::string courseName; // course name for this grade book
164      std::array<std::array<int, tests>, students> grades; // 2D array
165   };
```

Fig. 7.19 | Definition of class GradeBook that uses a two-dimensional array to store test grades. (Part 10 of 10.)

```
1  // Fig. 7.20: fig07_20.cpp
2  // Creates GradeBook object using a two-dimensional array of grades.
3  #include <array>
4  #include "GradeBook.h" // GradeBook class definition
5  using namespace std;
6
7  int main() {
8     // two-dimensional array of student grades
9     array<array<int, GradeBook::tests>, GradeBook::students> grades{
10       {87, 96, 70},
11       {68, 87, 90},
12       {94, 100, 90},
13       {100, 81, 82},
14       {83, 65, 85},
15       {78, 87, 65},
16       {85, 75, 83},
17       {91, 94, 100},
18       {76, 72, 84},
19       {87, 93, 73}};
```

Fig. 7.20 | Creates a `GradeBook` object using a two-dimensional `array` of grades, then invokes member function `processGrades` to analyze them. (Part 1 of 2.)

```
20
21      GradeBook myGradeBook("CS101 Introduction to C++ Programming", grades);
22      myGradeBook.displayMessage();
23      myGradeBook.processGrades();
24  }
```

Fig. 7.20 | Creates a `GradeBook` object using a two-dimensional `array` of grades, then invokes member function `processGrades` to analyze them. (Part 2 of 2.)

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

- C++ Standard Library class template `vector` is similar to class template `array`, but also supports dynamic resizing.
- Except for the features that modify a `vector`, the other features shown in Fig. 7.21 also work for `arrays`.
- Standard class template `vector` is defined in header `<vector>` (line 5) and belongs to namespace `std`.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 7.21: fig07_21.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>
6  #include <stdexcept>
7  using namespace std;
8
9  void outputVector(const vector<int>&); // display the vector
10 void inputVector(vector<int>&); // input values into the vector
11
12 int main() {
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:";
19    outputVector(integers1);
20
21    // print integers2 size and contents
22    cout << "\nSize of vector integers2 is " << integers2.size()
23       << "\nvector after initialization:";
24    outputVector(integers2);
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 1 of 7.)

```
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:";
33      outputVector(integers1);
34      cout << "integers2:";
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      }
43
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 2 of 7.)

```
44      // create vector integers3 using integers1 as an
45      // initializer; print size and contents
46      vector<int> integers3{integers1}; // copy constructor
47
48      cout << "\nSize of vector integers3 is " << integers3.size()
49         << "\nvector after initialization: ";
50      outputVector(integers3);
51
52      // use overloaded assignment (=) operator
53      cout << "\nAssigning integers2 to integers1:" << endl;
54      integers1 = integers2; // assign integers2 to integers1
55
56      cout << "integers1: ";
57      outputVector(integers1);
58      cout << "integers2: ";
59      outputVector(integers2);
60
61      // use equality (==) operator with vector objects
62      cout << "\nEvaluating: integers1 == integers2" << endl;
63
64      if (integers1 == integers2) {
65         cout << "integers1 and integers2 are equal" << endl;
66      }
67
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 3 of 7.)

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

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 4 of 7.)

```
86       // changing the size of a vector
87       cout << "\nCurrent integers3 size is: " << integers3.size() << endl;
88       integers3.push_back(1000); // add 1000 to the end of the vector
89       cout << "New integers3 size is: " << integers3.size() << endl;
90       cout << "integers3 now contains: ";
91       outputVector(integers3);
92    }
93
94    // output vector contents
95    void outputVector(const vector<int>& items) {
96       for (int item : items) {
97          cout << item << " ";
98       }
99
100      cout << endl;
101   }
102
103   // input vector contents
104   void inputVector(vector<int>& items) {
105      for (int& item : items) {
106         cin >> item;
107      }
108   }
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 5 of 7.)

```
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

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
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 6 of 7.)

```
Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

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

Current integers3 size is: 7
New integers3 size is: 8
integers3 now contains: 1 2 3 4 5 6 7 1000
```

Fig. 7.21 | Demonstrating C++ Standard Library class template `vector`. (Part 7 of 7.)

7.10 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 most data types.
- `vector` member function `size` obtain the number of elements in the `vector`.
- As with class template `array`, you can also do this using a counter-controlled loop and the subscript ([ ]) operator.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

- Notice that we used parentheses rather than braces to pass the size argument to each `vector` object's constructor.
- When creating a `vector`, if the braces contain one value of the `vector`'s element type, the braces are treated as a one-element initializer list, rather than a call to the constructor that sets the `vector`'s size.
- The following declaration actually creates a one-element `vector<int>` containing the `int` value 7, not a 7-element `vector`
  - `vector<int> integers1{7};`

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

- You can use the assignment (=) operator with `vector` objects.
- As is the case with `arrays`, C++ is not required to perform bounds checking when `vector` elements are accessed with square brackets.
- Standard class template `vector` provides bounds checking in its member function `at` (as does class template `array`).

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

- An exception indicates a problem that occurs while a program executes.
- The name "exception" suggests that the problem occurs infrequently.
- Exception handling enables you to create fault-tolerant programs that can resolve (or handle) exceptions.
- When a function detects a problem, such as an invalid `array` subscript or an invalid argument, it throws an exception—that is, an exception occurs.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

- To handle an exception, place any code that might throw an exception in a `try` statement.
- The `try` block contains the code that might throw an exception, and the `catch` block contains the code that handles the exception if one occurs.
- You can have many `catch` blocks to handle different types of exceptions that might be thrown in the corresponding `try` block.
- The `vector` member function `at` provides bounds checking and throws an exception if its argument is an invalid subscript.
- By default, this causes a C++ program to terminate.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

Performance Tip 7.2

Catching an exception by reference increases performance by preventing the exception object from being copied when it's caught. You'll see in later chapters that catching by reference is also important when defining `catch` *blocks that process related exception types.*

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

Changing the Size of a `vector`

- One of the key differences between a `vector` and an `array` is that a `vector` can dynamically grow and shrink as the number of elements it needs to accommodate varies.
- To demonstrate this, line 88 shows the current size of `integers3`, line 89 calls the `vector`'s `push_back` member function to add a new element containing 1000 to the end of the `vector` and line 90 shows the new size of `integers3`.
- Line 92 then displays `integers3`'s new contents.

©1992–2014 by Pearson Education, Inc. All Rights Reserved.

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

C++11: List Initializing a `vector`

- Many of the `array` examples in this chapter used list initializers to specify the initial `array` element values.
- C++11 also allows this for `vectors` (and other C++ Standard Library data structures).

©1992–2014 by Pearson Education, Inc. All Rights Reserved.