Topic 4
Arrays and Pointers
Lecture 4a (chapter 7)

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

Agenda

- Arrays
- Using class template *array*
- Declare arrays with initialization
- Use arrays to store sort and lists and tables of values
- Use range-based for statement
- Pass arrays to functions
- Sort arrays
- Binary search of sorted array
- Multidimensional arrays
- std::vector

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.

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).

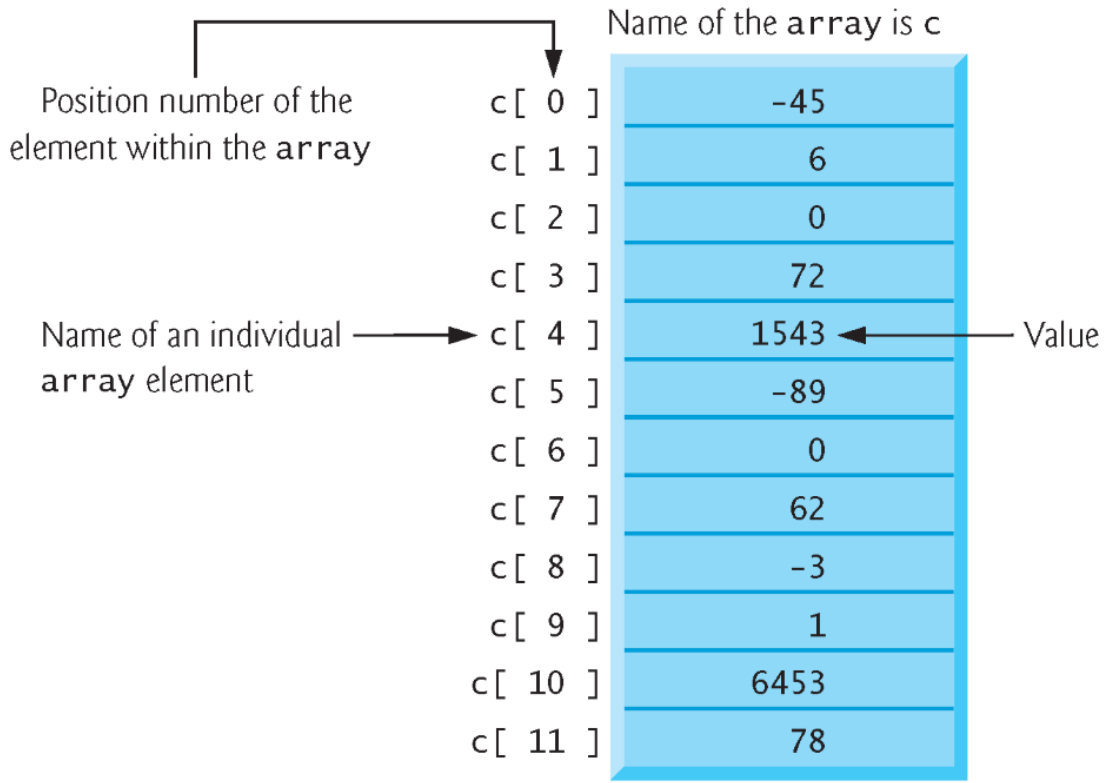

Fig. 7.1 | array of 12 elements.

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.

Standard C-type arrays

- The concept of an array of data has been around for a long time
- The simplest way to declare an array is as an array-type variable with the size of the array.
- Example: an array of 50 integers:

```
int values[50];
```

- With the above sample, a contiguous space in memory is created that is large enough to store 50 integers (if an *int* is 4 bytes then 200 bytes are reserved).
- This space is reserved on the stack in the activation record of the function in which this statement exists.
- This type of array is not an object. There is no means of finding its size. If you pass the array to a function for processing, you must also pass its size as well.

7.3 Declaring arrays using the array class

- C++ has an *array* class template in the standard template library.
- For most applications, this class offers the same performance as C-type arrays.
- To use it you must include the array header file (#include <array>)
- 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.

7.4 Examples Using arrays

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. In GCC it usually represents an unsigned long.
- 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.

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 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.

Figure 4.7

```
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.
  - You should not use literal values, always assign such a value as a constant.
- Constant variables are also called named constants or read-only variables.

```
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.)

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.*

7.4.6 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.

```
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.)

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.*

7.5 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.

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] << " ";
}
```

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.

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.
- Sorting data—placing it into ascending or descending order—is one of the most important computing applications.
- Often it may be necessary to determine whether an array contains a value that matches a certain key value.
- This is called 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.

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.

```
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.

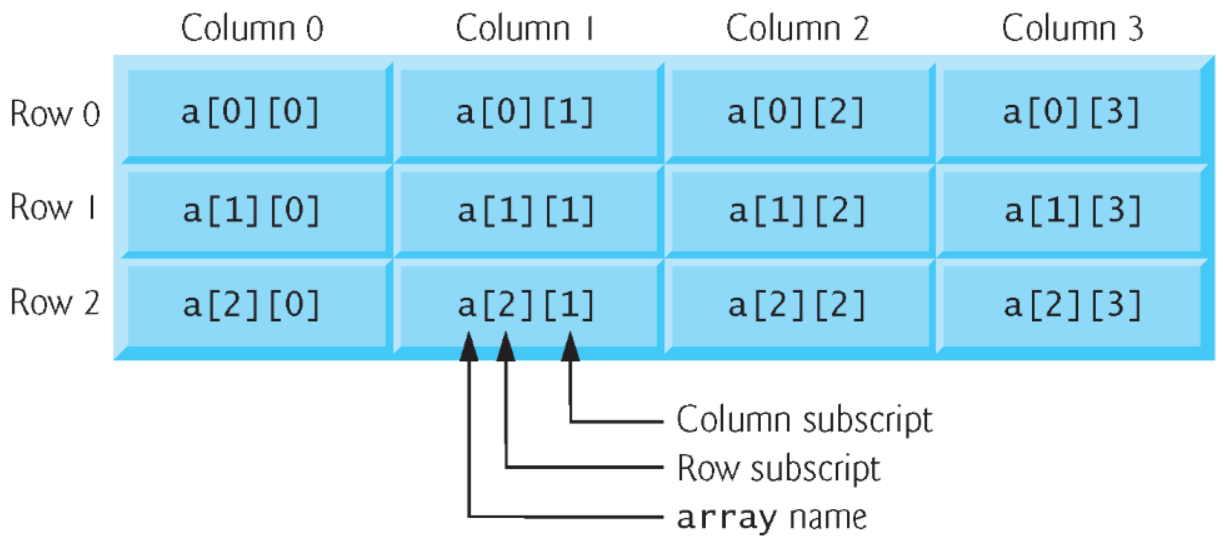

Fig. 7.16 | Two-dimensional `array` with three rows and four columns.

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.

```
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.

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;
}
```

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.

```
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.
- vectors 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.

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 creates a one-element vector<int> containing the int value 7, not a 7-element vector

```
vector<int> integers1{7};
```

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).**

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.
- Include <stdexcept> or <exception> headers.

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.

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.

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).