```
1  // Fig. 7.9: fig07_09.cpp
2  // Bar chart printing program.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10    const size_t arraySize = 11;
11    array< unsigned int, arraySize > n =
12       { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
13
14    cout << "Grade distribution:" << endl;
15
```

Fig. 7.9 | Bar chart printing program. (Part 1 of 3.)

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

Fig. 7.9 | 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. 7.9 | 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 6,000,000 times.
- An `array` version of this program is shown in Fig. 7.10.
- This version also uses the new C++11 random-number generation capabilities that were introduced in Section 6.9.
- *The single statement in line 22 of this program replaces the* `switch` *statement in lines 23–45 of Fig. 6.9.*

```
1  // Fig. 7.10: fig07_10.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 {
12    // use the default random-number generation engine to
13    // produce uniformly distributed pseudorandom int values from 1 to 6
14    default_random_engine engine( static_cast< unsigned int >( time(0) ) );
15    uniform_int_distribution< unsigned int > randomInt( 1, 6 );
16
17    const size_t arraySize = 7; // ignore element zero
18    array< unsigned int, arraySize > frequency = {}; // initialize to 0s
19
20    // roll die 6,000,000 times; use die value as frequency index
21    for ( unsigned int roll = 1; roll <= 6000000; ++roll )
22       ++frequency[ randomInt( engine ) ];
23
```

Fig. 7.10 | 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 ]
29            << endl;
30   } // end main
```

```
Face    Frequency
   1      1000167
   2      1000149
   3      1000152
   4       998748
   5       999626
   6      1001158
```

Fig. 7.10 | Die-rolling program using an `array` instead of `switch`. (Part 2 of 2.)

7.4.7 Using `arrays` to Summarize Survey Results

- Our next example (Fig. 7.11) 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.
- In Section 7.10, we demonstrate the class template `vector`'s at function, which performs bounds checking for you.
- Class template `array` also has an `at` function.

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

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

```
21      // for each answer, select responses element and use that value
22      // as frequency subscript to determine element to increment
23      for ( size_t answer = 0; answer < responses.size(); ++answer )
24         ++frequency[ responses[ answer ] ];
25
26      cout << "Rating" << setw( 17 ) << "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( 17 ) << frequency[ rating ]
31            << endl;
32   } // end main
```

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

Fig. 7.11 | 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 `array`s 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.

Common Programming Error 7.4

Referring to an element outside the `array` bounds is an execution-time logic error. It isn't 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 Chapters 15–16, you'll learn about iterators, 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.

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. 7.12: fig07_12.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 {
13    cout << "First call to each function:\n";
14    staticArrayInit();
15    automaticArrayInit();
16
17    cout << "\n\nSecond call to each function:\n";
18    staticArrayInit();
19    automaticArrayInit();
20    cout << endl;
21 } // end main
22
```

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