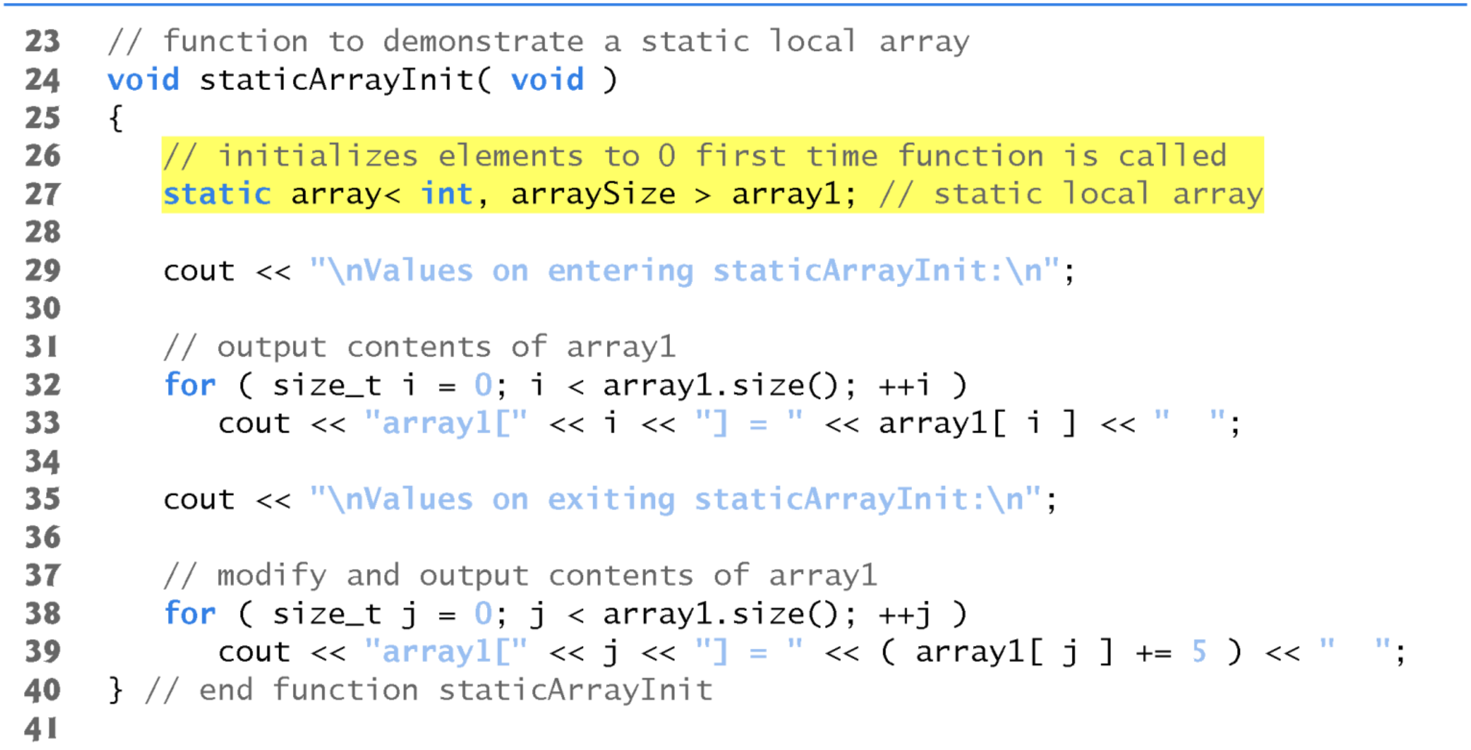


```cpp
// function to demonstrate a static local array
void staticArrayInit( void )
{
   // initializes elements to 0 first time function is called
   static array< int, arraySize > array1; // static local array

   cout << "\nValues on entering staticArrayInit:\n";

   // output contents of array1
   for ( size_t i = 0; i < array1.size(); ++i )
      cout << "array1[" << i << "] = " << array1[ i ] << "  ";

   cout << "\nValues on exiting staticArrayInit:\n";

   // modify and output contents of array1
   for ( size_t j = 0; j < array1.size(); ++j )
      cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "  ";
} // end function staticArrayInit

```

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

```cpp
// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
   // initializes elements each time function is called
   array< int, arraySize > array2 = { 1, 2, 3 }; // automatic local array

   cout << "\n\nValues on entering automaticArrayInit:\n";

   // output contents of array2
   for ( size_t i = 0; i < array2.size(); ++i )
      cout << "array2[" << i << "] = " << array2[ i ] << "  ";

   cout << "\nValues on exiting automaticArrayInit:\n";

   // modify and output contents of array2
   for ( size_t j = 0; j < array2.size(); ++j )
      cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "  ";
} // end function automaticArrayInit
```

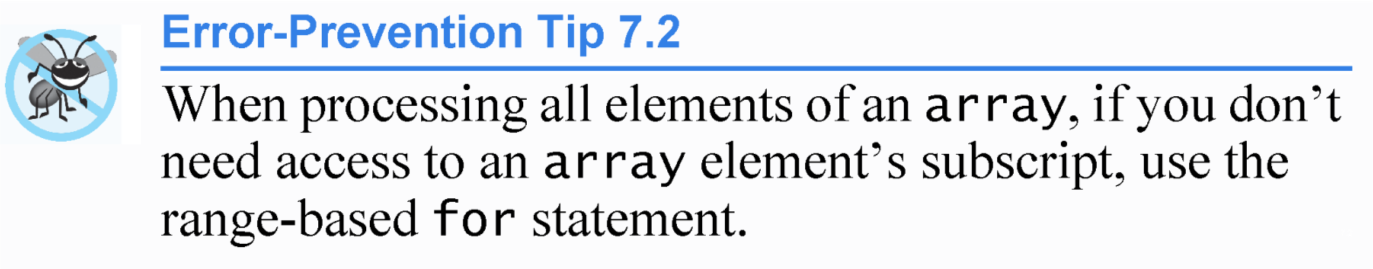
Fig. 7.12 | static array initialization and automatic array initialization. (Part 3 of 4.)

```
First call to each function:

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

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

Second call to each function:

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

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

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

7.5 Range-Based `for` Statement

- It’s common to process *all* the elements of an `array`.
- The new 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.



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.

7.5 Range-Based `for` Statement (cont.)

- The syntax of a range-based for statement is:

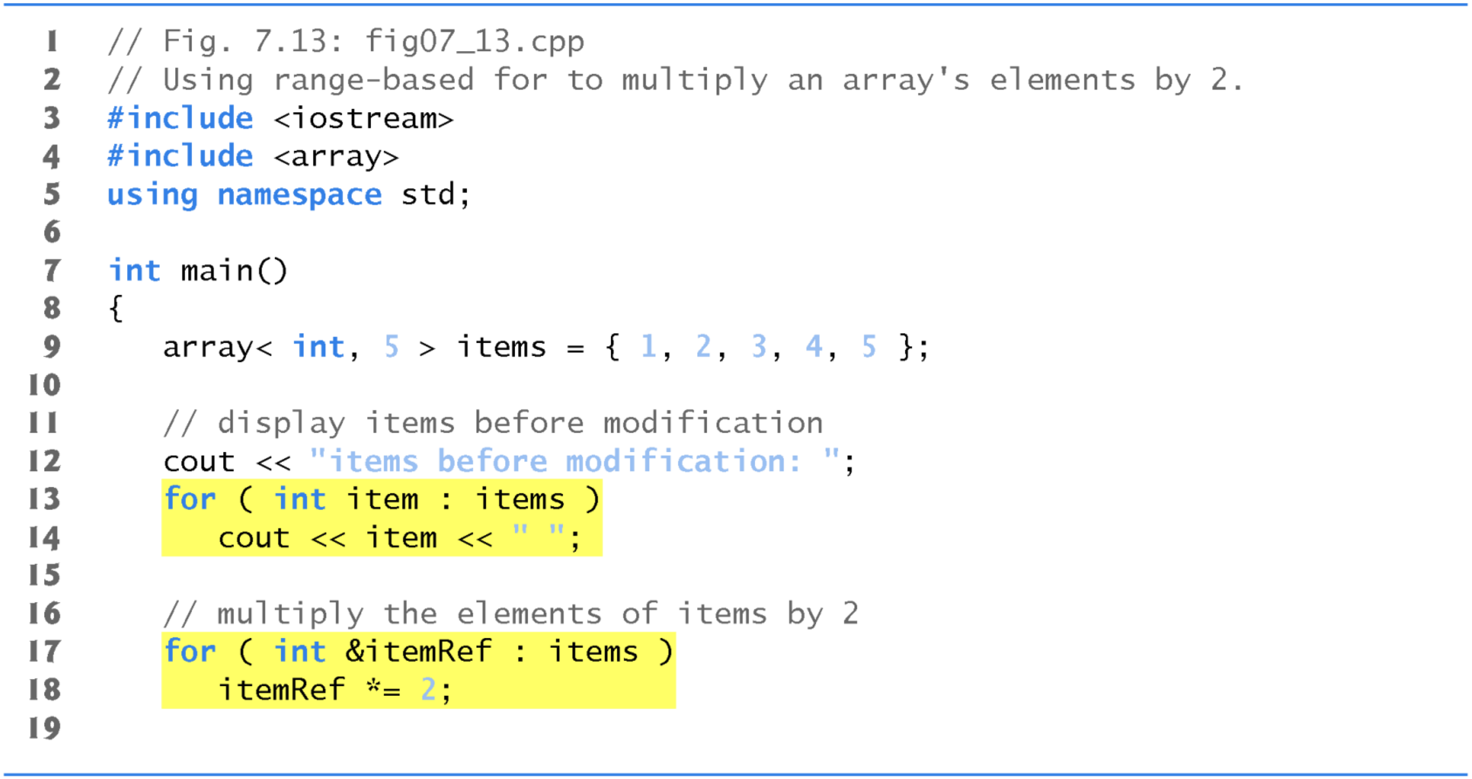
  `for (` *rangeVariableDeclaration* `:` *expression* `)`

  statement

- where *rangeVariableDeclaration* has a type and an identifier (e.g., `int item`), and *expression* is the `array` through which to iterate.
- The type in the *rangeVariableDeclaration* must be *consistent* with the type of the `array`'s elements.

7.5 Range-Based `for` Statement (cont.)

- You can use the range-based `for` statement with most of the C++ Standard Library's prebuilt data structures (commonly called *containers*), including classes `array` and `vector`.
- Figure 7.13 uses the range-based `for` to display an `array`'s contents (lines 13–14 and 22–23) and to multiply each of the `array`'s element values by 2 (lines 17–18).



```cpp
// Fig. 7.13: fig07_13.cpp
// Using range-based for to multiply an array's elements by 2.
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array< int, 5 > items = { 1, 2, 3, 4, 5 };

   // display items before modification
   cout << "items before modification: ";
   for ( int item : items )
      cout << item << " ";

   // multiply the elements of items by 2
   for ( int &itemRef : items )
      itemRef *= 2;

```

Fig. 7.13 | Using range-based `for` to multiply an `array`'s elements by 2. (Part 1 of 2.)

```cpp
   // display items after modification
   cout << "\nitems after modification: ";
   for ( int item : items )
      cout << item << " ";

   cout << endl;
} // end main
```

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

Fig. 7.13 | 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 13 can be read as "for each iteration, assign the next element of items to int variable item, then execute the following statement."
- Lines 13–14 are equivalent to the following counter-controlled repetition:

```
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 17–18 use a range-based `for` statement to multiply each element of `items` by 2.
- In line 17, the *rangeVariableDeclaration* 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

7.5 Range-Based `for` Statement (cont.)

Using an Element’s Subscript

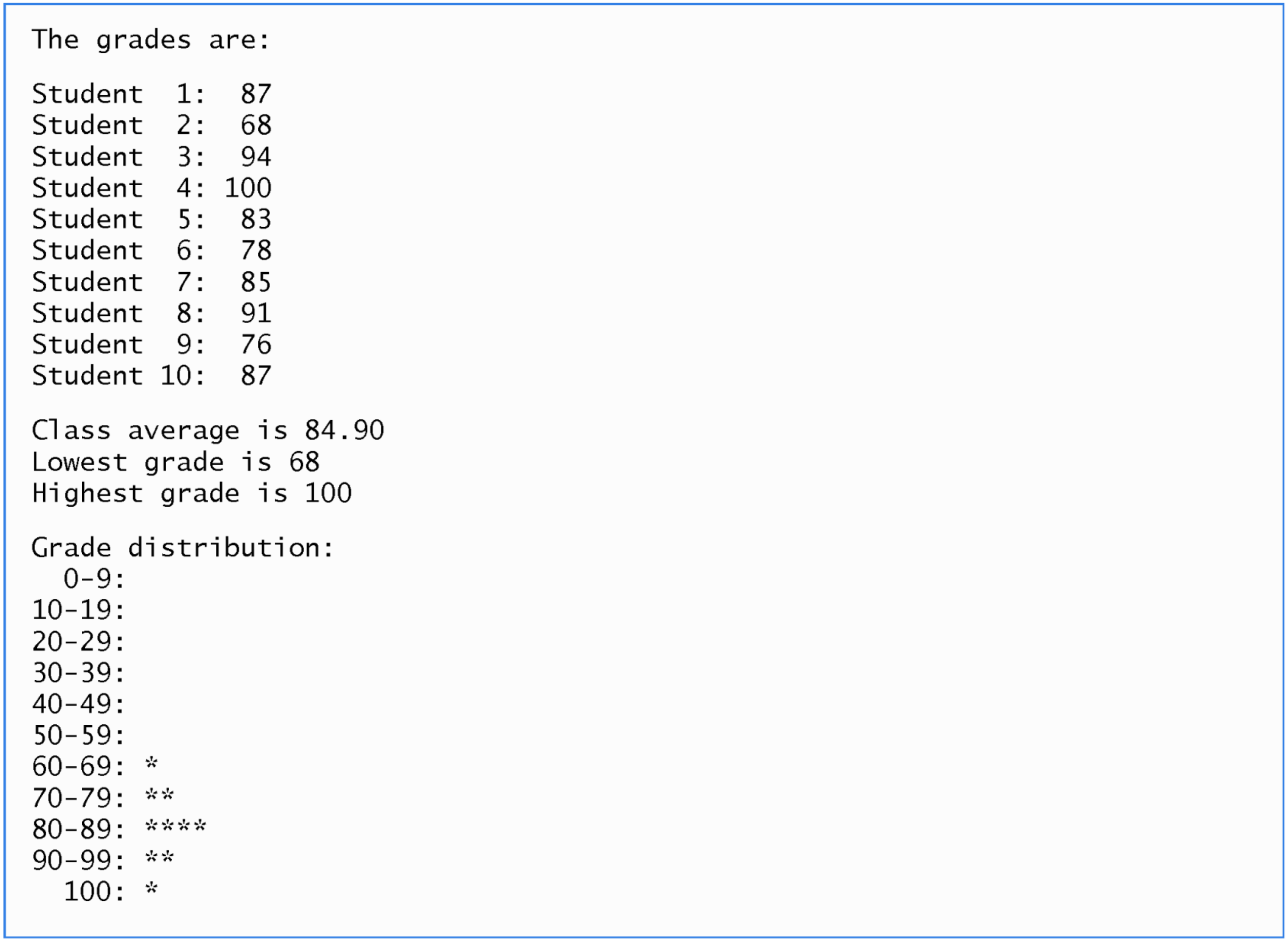
- The range-based `for` statement can be used in place of the counter-controlled `for` statement whenever code looping through an array does not require access to the element’s subscript.
- However, if a program must use subscripts for some reason other than simply to loop through an `array` (e.g., to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

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

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

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

Fig. 7.14 | Output of the `GradeBook` example that stores grades in an `array`. (Part 1 of 2.)



```
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

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