Computer Programming
Lecture 8

Hung-Yu Wei
Department of Electrical Engineering
National Taiwan University

Review

- Pass parameters to a function
  - Pass-by-value
  - Pass-by-reference
- How should we pass an array to a function?

6.5 Passing Arrays to Functions

- Specify name without []
  - To pass array **`myArray`** to **`myFunction`**

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

Array size=24

  - Array size is usually passed (but not required)
    - Useful to compute all elements in the array

Passing Arrays to Functions

- Arrays passed-by-reference
  - Functions can modify original array data
  - Value of name of array is address of first element
    - Function knows where the array is stored
    - Can change original memory locations

- Individual array elements passed-by-value
  - Like regular variables
  - `square( myArray[3] );`

Passing Arrays to Functions

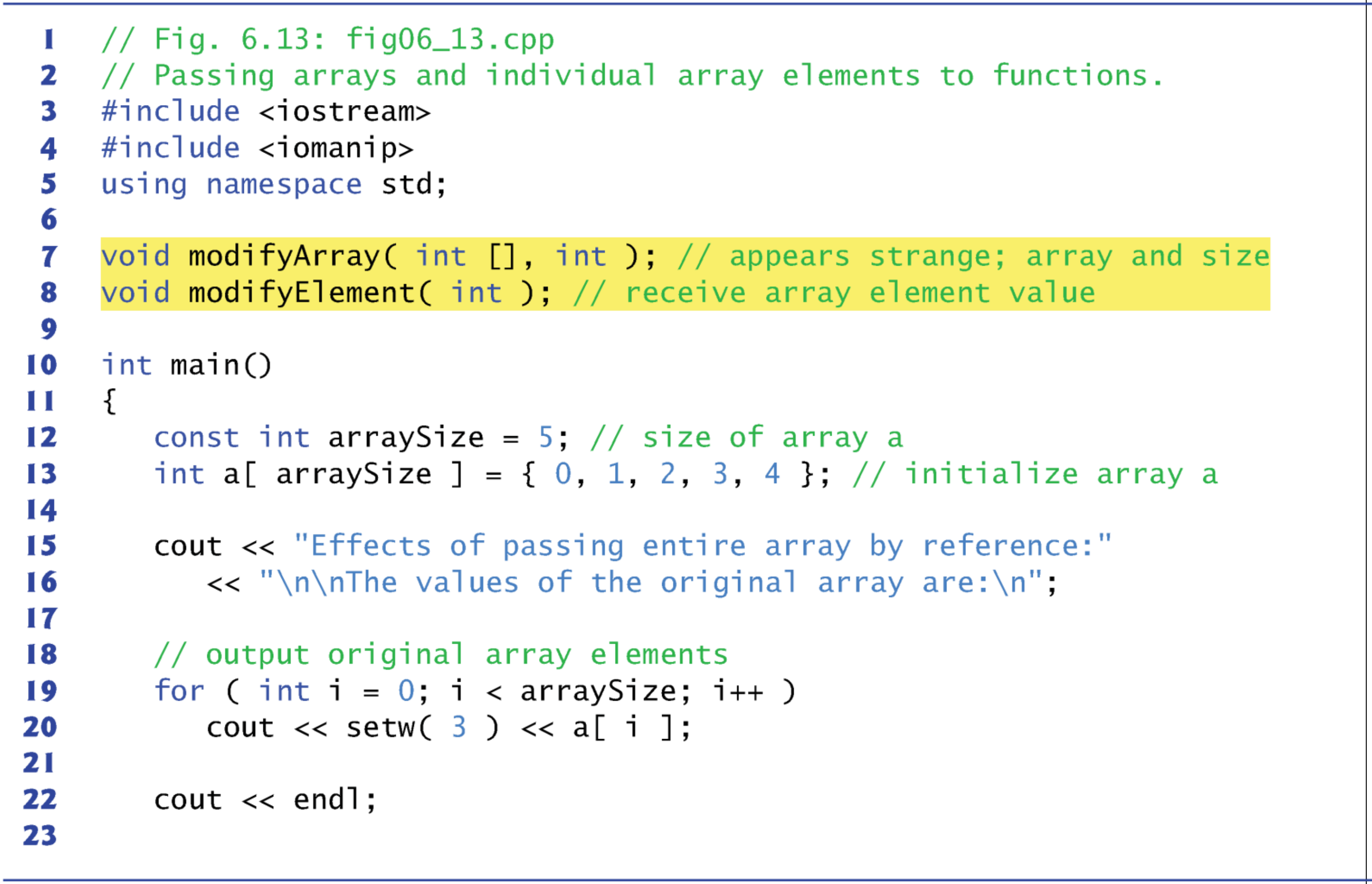
- Functions taking arrays
  - Function prototype
    - `void modifyArray( int b[], int arraySize );`
    - `void modifyArray( int [], int );`
      - Names optional in prototype (just like what we did before)
    - Both take an integer array and a single integer
  - Array size between brackets is not needed
    - Ignored by compiler
  - If array is declared as **`const`**
    - Cannot be modified (compiler error if you try to modify it)
    - `void doNotModify( const int [] );`

Example: array and function

-
  ```
  void myFunc(double[],int);   //function prototype
  ```

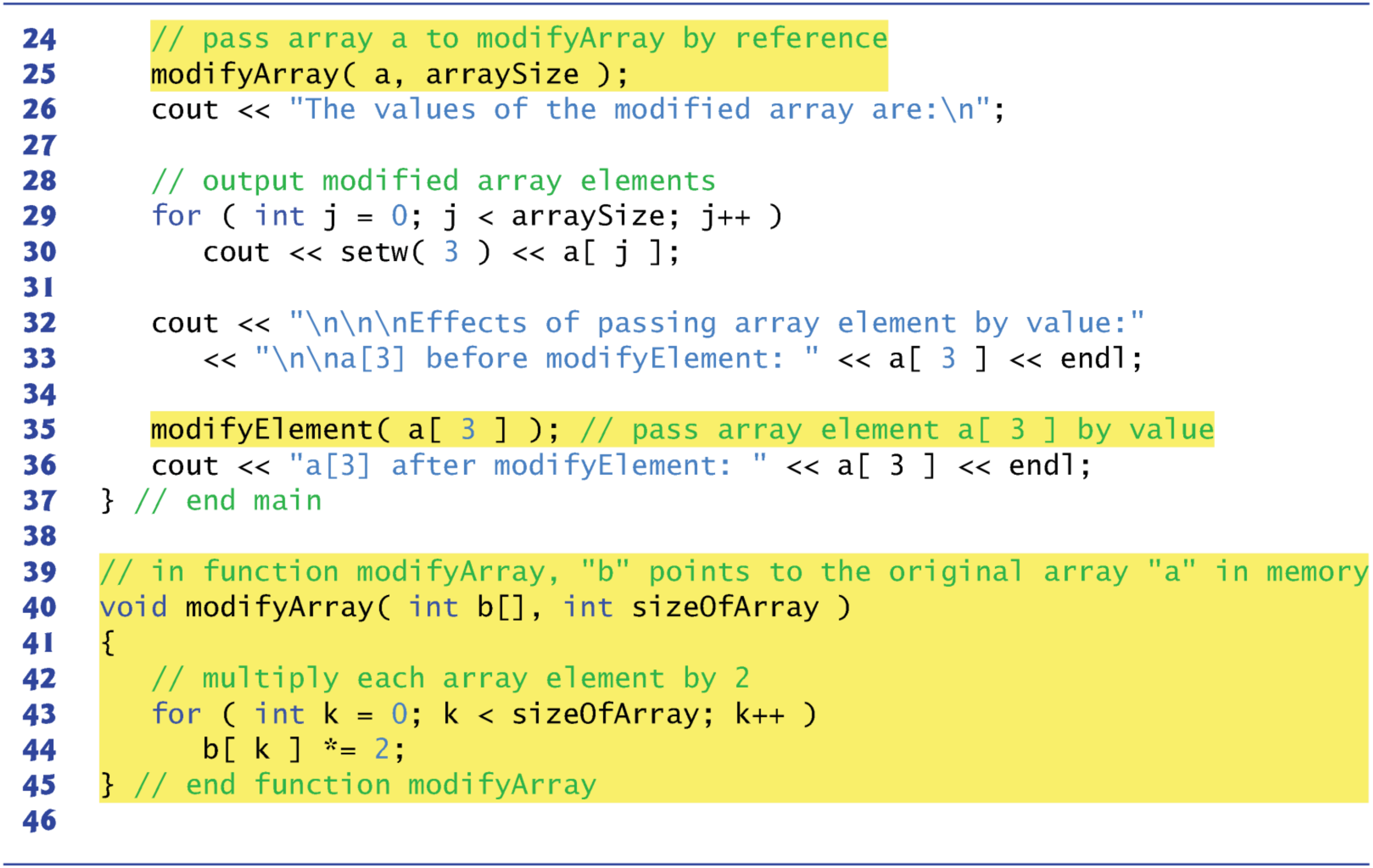
-
  ```
  int main()
  {
          int n=3;
          double A[n]={1,3,5};
          myFunc(A,n);                //call the function
          …
  }
  ```

-
  ```
  void myFunc(double X[],int Y)   //implementation
  {
          for (int i=0; i<Y; i++)
                  X[i]++;
  }
  ```



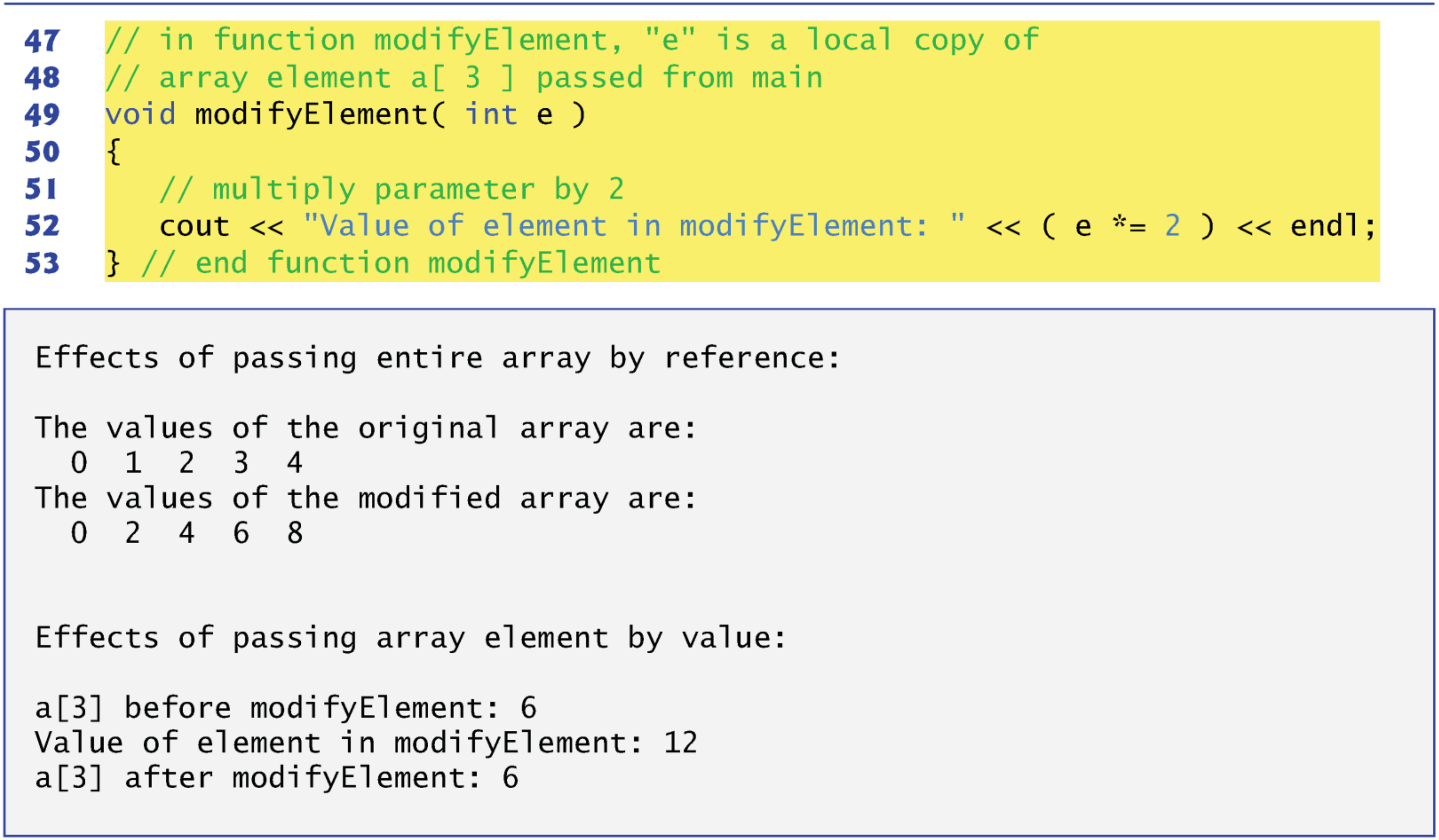
```
// Fig. 6.13: fig06_13.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>
#include <iomanip>
using namespace std;

void modifyArray( int [], int ); // appears strange; array and size
void modifyElement( int ); // receive array element value

int main()
{
   const int arraySize = 5; // size of array a
   int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a

   cout << "Effects of passing entire array by reference:"
      << "\n\nThe values of the original array are:\n";

   // output original array elements
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];

   cout << endl;

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 1 of 3.)



```cpp
   // pass array a to modifyArray by reference
   modifyArray( a, arraySize );
   cout << "The values of the modified array are:\n";

   // output modified array elements
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 3 ) << a[ j ];

   cout << "\n\n\nEffects of passing array element by value:"
      << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;

   modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
   cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
} // end main

// in function modifyArray, "b" points to the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; k++ )
      b[ k ] *= 2;
} // end function modifyArray

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 3.)



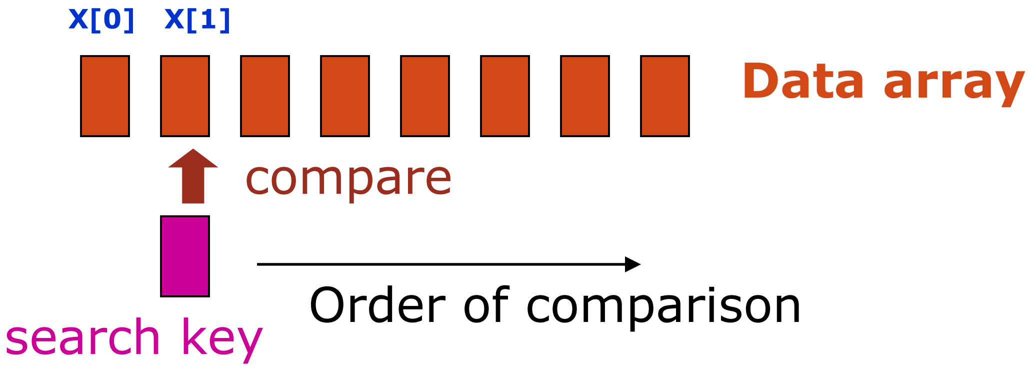
```cpp
// in function modifyElement, "e" is a local copy of
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
} // end function modifyElement
```

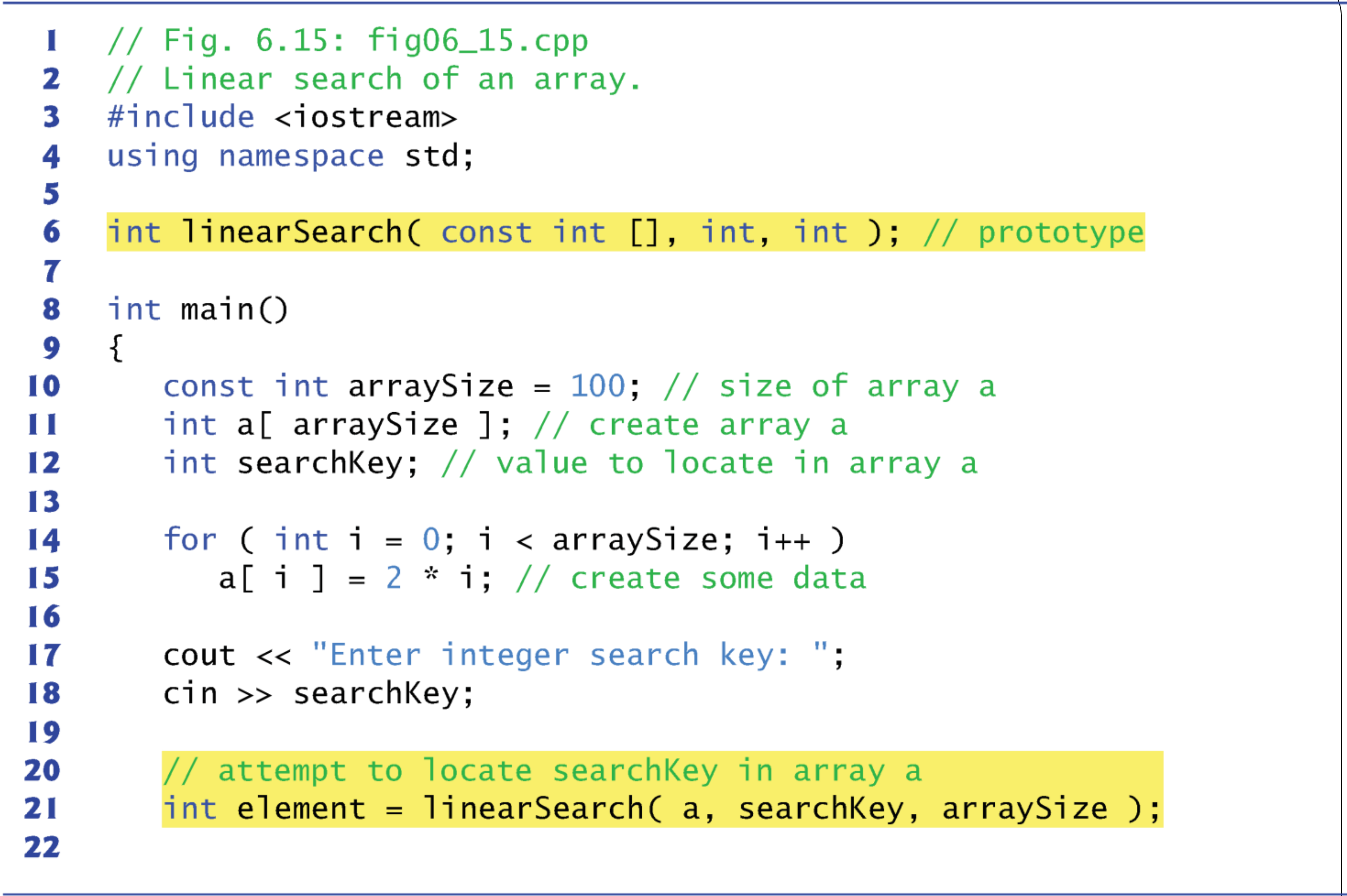
```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 3 of 3.)

6.6 Searching Within a Array

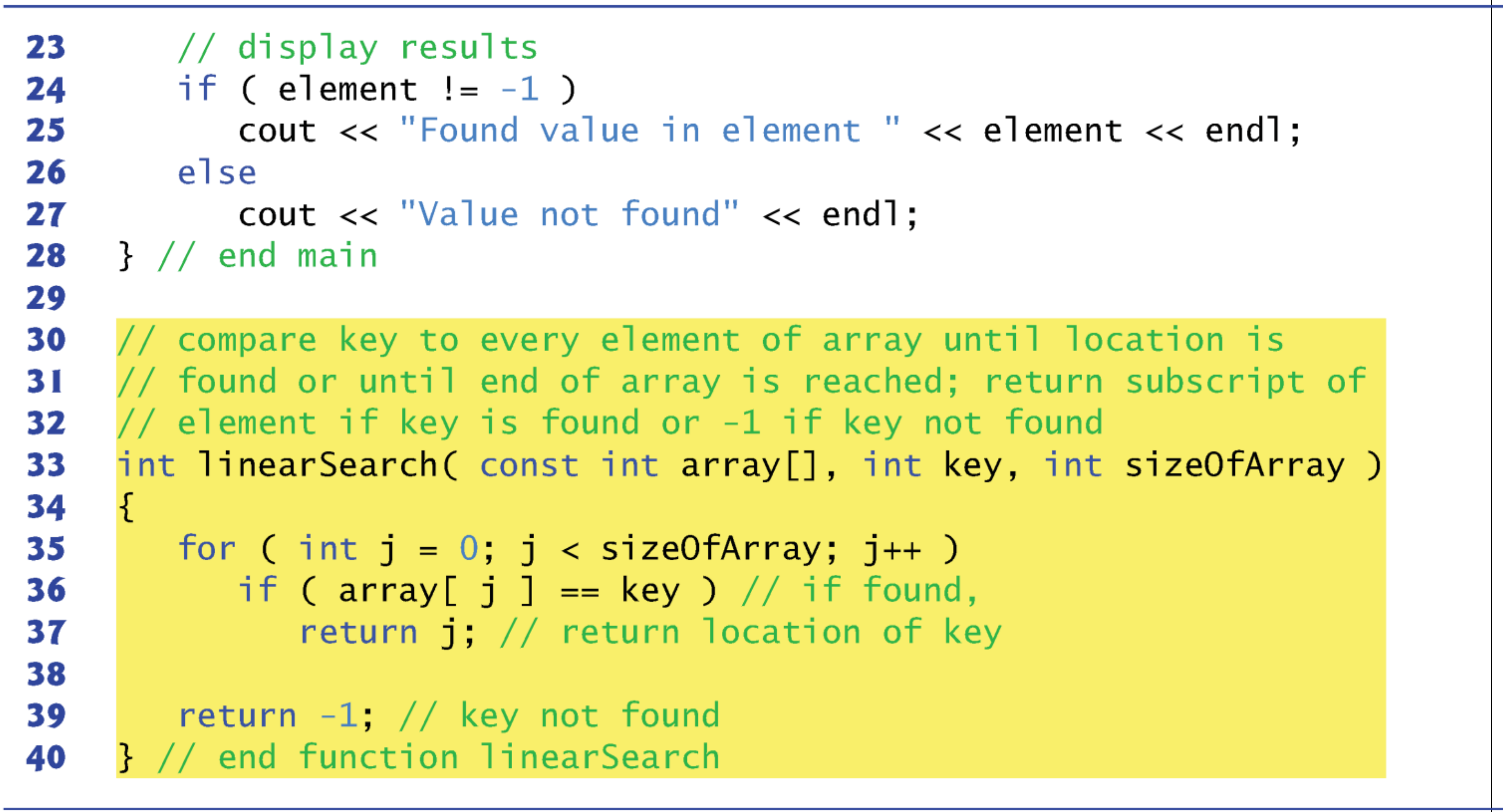
- Linear Search
  - Compare each element of an array with a search key
  - Useful for small and unsorted arrays
    - Inefficient. If search key not present, examine every element





```
// Fig. 6.15: fig06_15.cpp
// Linear search of an array.
#include <iostream>
using namespace std;

int linearSearch( const int [], int, int ); // prototype

int main()
{
   const int arraySize = 100; // size of array a
   int a[ arraySize ]; // create array a
   int searchKey; // value to locate in array a

   for ( int i = 0; i < arraySize; i++ )
      a[ i ] = 2 * i; // create some data

   cout << "Enter integer search key: ";
   cin >> searchKey;

   // attempt to locate searchKey in array a
   int element = linearSearch( a, searchKey, arraySize );

```

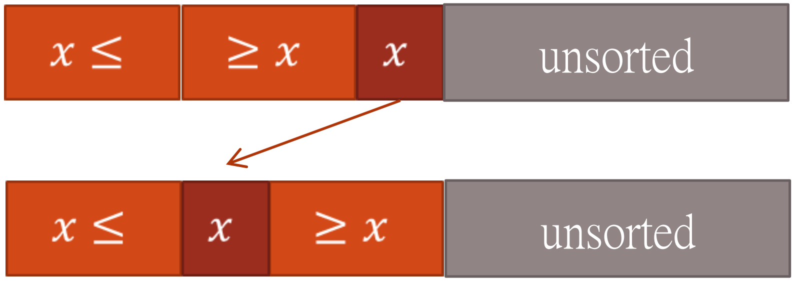
Fig. 6.15 | Linear search of an array. (Part 1 of 3.)



```cpp
   // display results
   if ( element != -1 )
      cout << "Found value in element " << element << endl;
   else
      cout << "Value not found" << endl;
} // end main

// compare key to every element of array until location is
// found or until end of array is reached; return subscript of
// element if key is found or -1 if key not found
int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; j++ )
      if ( array[ j ] == key ) // if found,
         return j; // return location of key

   return -1; // key not found
} // end function linearSearch
```

Fig. 6.15 | Linear search of an array. (Part 2 of 3.)

6.7 Sorting (Insertion Sort)



- Insertion Sort
  - Simple but inefficient
- In the *x*-th iteration
  - Goal: sort the first *x* elements
    - Insert the x-th element to the "right place"
      - This is called "Insertion Sort"
  - Iterative Step: compare the x-th element with k-th element (k= x-1, x-2, ….)
    - Insert the x-th element to the correct position
- There are other sorting algorithms

Example: Insertion Sort

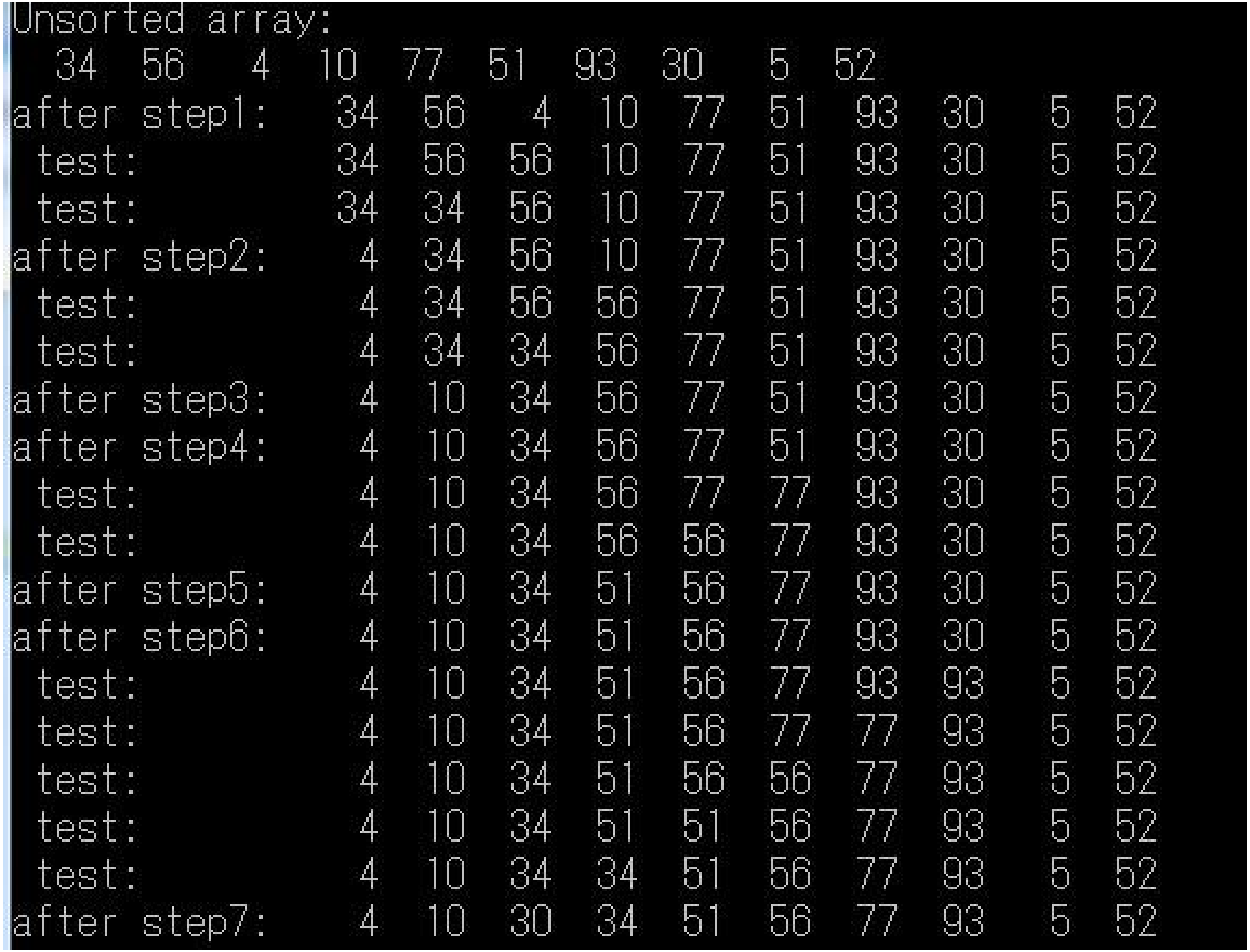
- Initial Value

  34 56 4 10 77 51 …
- 1st Iteration

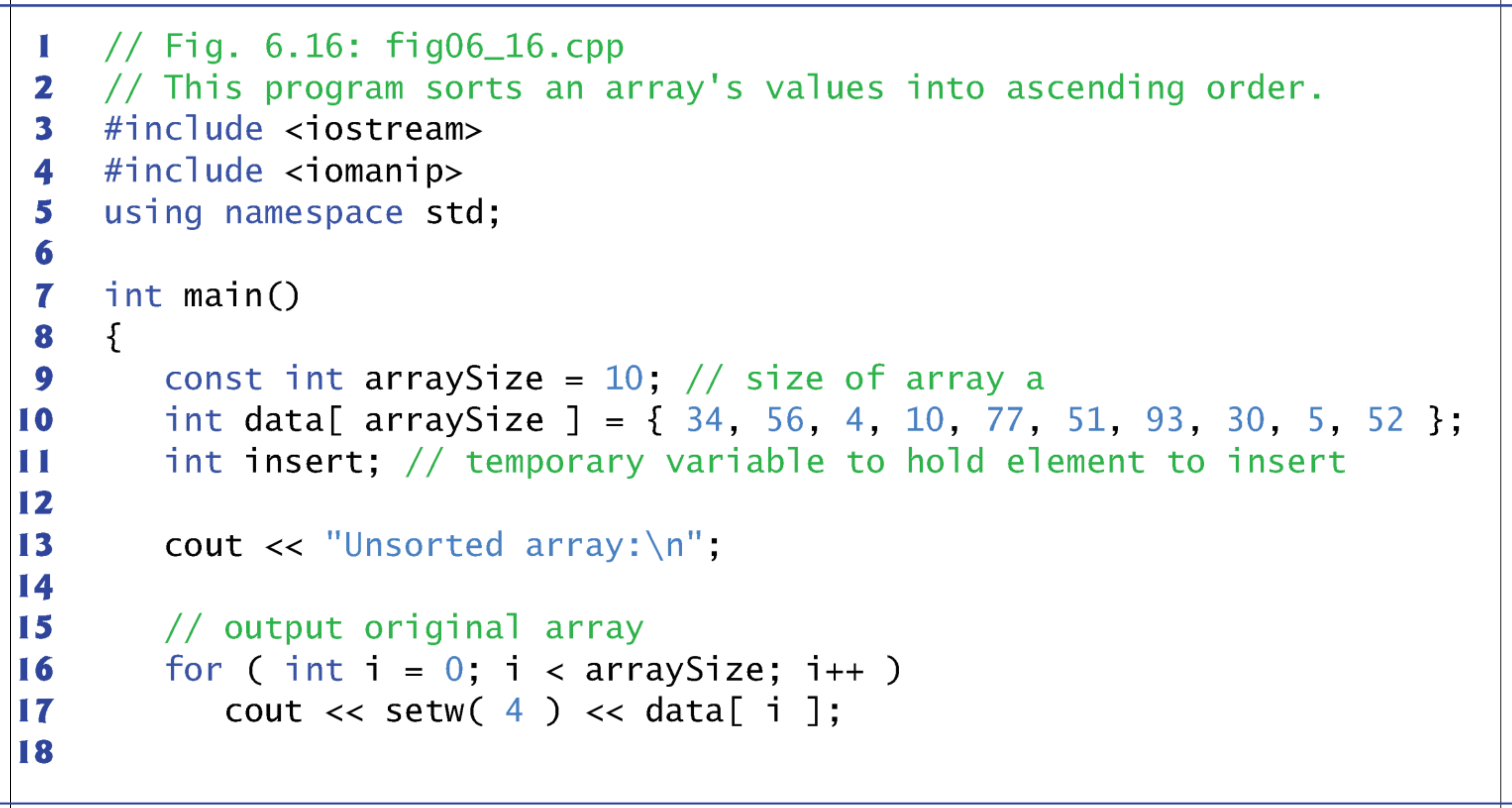
  34 56 4 10 …
- 2nd Iteration

  4 34 56 10
- 3rd Iteration

  4 10 34 56 …

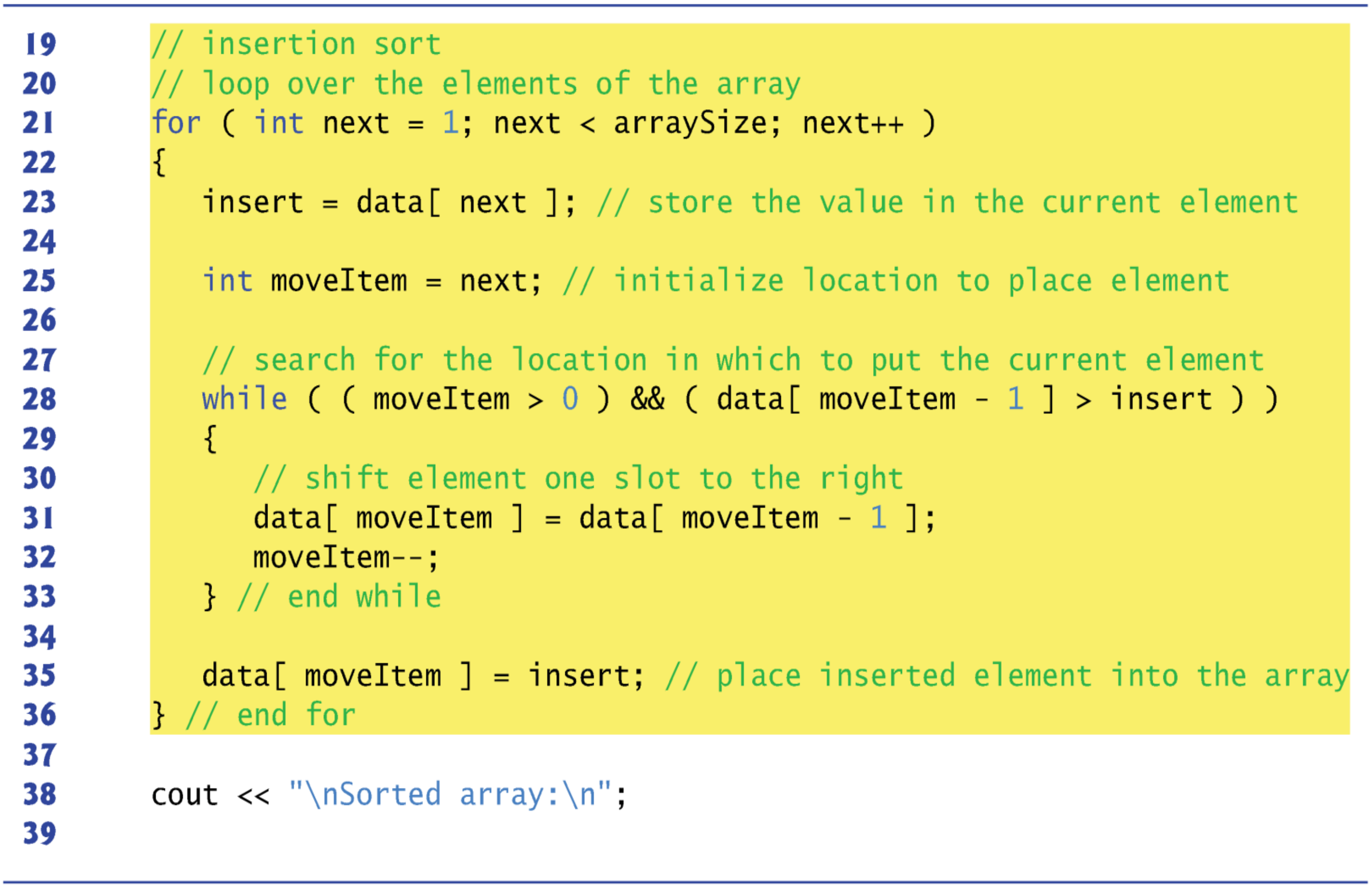


```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
after step1:   34   56    4   10   77   51   93   30    5   52
 test:         34   56   56   10   77   51   93   30    5   52
 test:         34   34   56   10   77   51   93   30    5   52
after step2:    4   34   56   10   77   51   93   30    5   52
 test:          4   34   56   56   77   51   93   30    5   52
 test:          4   34   34   56   77   51   93   30    5   52
after step3:    4   10   34   56   77   51   93   30    5   52
after step4:    4   10   34   56   77   51   93   30    5   52
 test:          4   10   34   56   77   77   93   30    5   52
 test:          4   10   34   56   56   77   93   30    5   52
after step5:    4   10   34   51   56   77   93   30    5   52
after step6:    4   10   34   51   56   77   93   30    5   52
 test:          4   10   34   51   56   77   93   93    5   52
 test:          4   10   34   51   56   77   77   93    5   52
 test:          4   10   34   51   56   56   77   93    5   52
 test:          4   10   34   51   51   56   77   93    5   52
 test:          4   10   34   34   51   56   77   93    5   52
after step7:    4   10   30   34   51   56   77   93    5   52
```



```cpp
// Fig. 6.16: fig06_16.cpp
// This program sorts an array's values into ascending order.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int arraySize = 10; // size of array a
   int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
   int insert; // temporary variable to hold element to insert

   cout << "Unsorted array:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

```

Fig. 6.16 | Sorting an array with insertion sort. (Part 1 of 3.)



```cpp
// insertion sort
// loop over the elements of the array
for ( int next = 1; next < arraySize; next++ )
{
   insert = data[ next ]; // store the value in the current element

   int moveItem = next; // initialize location to place element

   // search for the location in which to put the current element
   while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
   {
      // shift element one slot to the right
      data[ moveItem ] = data[ moveItem - 1 ];
      moveItem--;
   } // end while

   data[ moveItem ] = insert; // place inserted element into the array
} // end for

cout << "\nSorted array:\n";

```

Fig. 6.16 | Sorting an array with insertion sort. (Part 2 of 3.)

```
   // output sorted array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

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

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

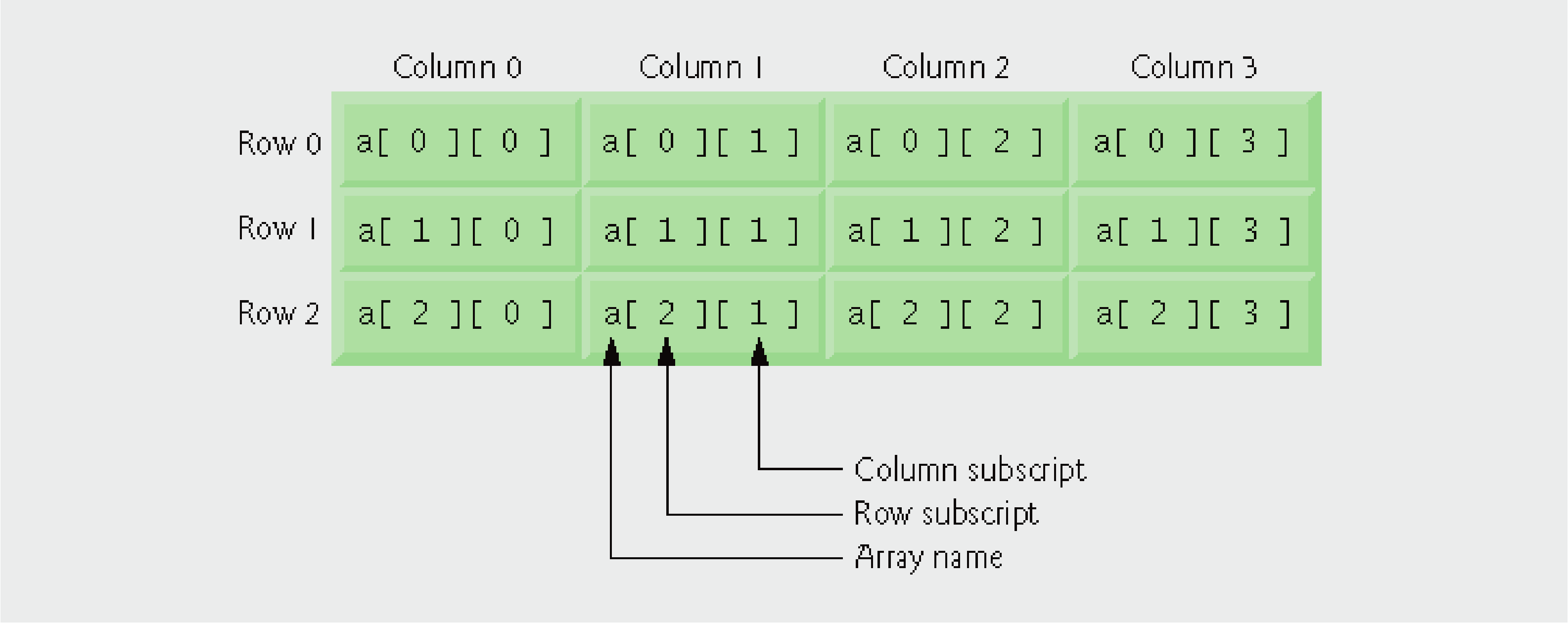
Fig. 6.16 | Sorting an array with insertion sort. (Part 3 of 3.)

6.8 Multidimensional Arrays

- Multiple subscripts
  - Two-Dimensional array (2-D array)
    - a[i][j]
  - Tables with rows and columns
  - Specify row, then column
    - a[i][j]
    - i-th row, j-th column

Multidimensional Arrays

- "Array of arrays"
  - **`a[0]`** is an array of 4 elements
  - **`a[0][0]`** is the first element of that array



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

2-D array initialization examples

- int a[2][2]={{1,2},{3,4}};

1	2
3	4

- int a[2][2]={{1},{3,4}};

Default value is 0

1	0
3	4

- int a[2][2]={1,2,3};

First Row 0, and then Row 1

1	2
3	0

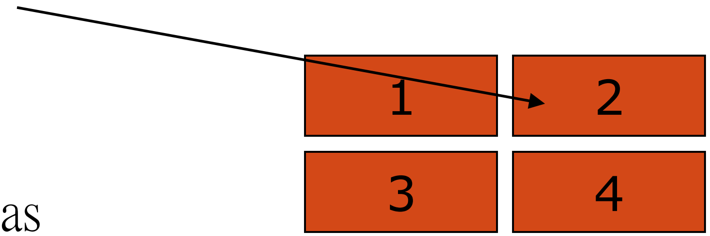
Array as input to a function

- Function prototypes
  - Must specify sizes of subscripts
    - First subscript not necessary (just like 1-D arrays)
    - **`void printArray( int [ ][ 3 ] );`**
  - This can also be used
    - **`void printArray( int [ 2 ][ 3 ] );`**

Use array elements

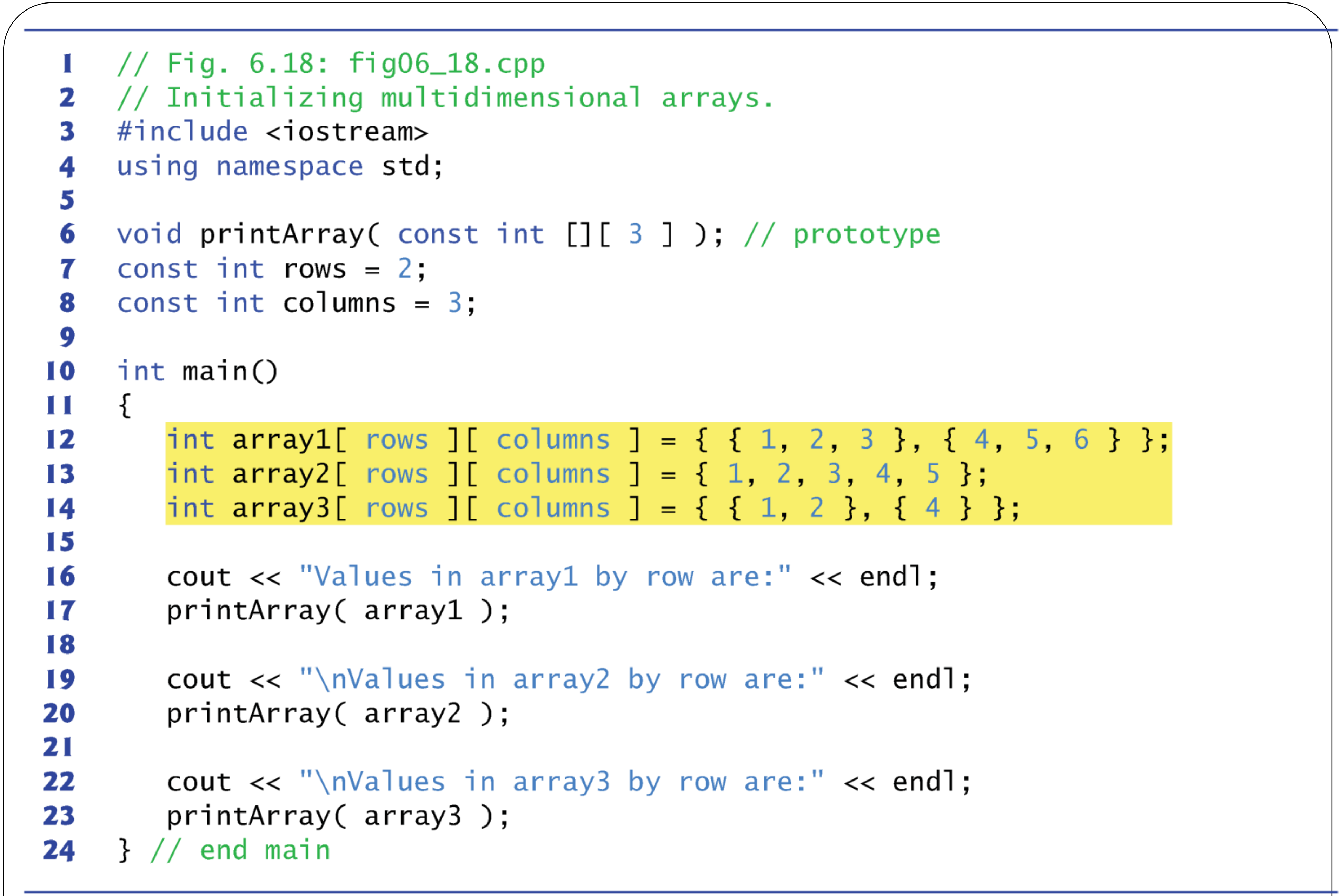
- Reference array elements
  - (Example)Referenced like normal variables

```
cout << b[ 0 ][ 1 ];
```



  - Display result: **2**
  - Syntax error
    - Cannot reference using commas

```
cout << b[ 0, 1 ];
```



```cpp
// Fig. 6.18: fig06_18.cpp
// Initializing multidimensional arrays.
#include <iostream>
using namespace std;

void printArray( const int [][ 3 ] ); // prototype
const int rows = 2;
const int columns = 3;

int main()
{
   int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ rows ][ columns ] = { 1, 2, 3, 4, 5 };
   int array3[ rows ][ columns ] = { { 1, 2 }, { 4 } };

   cout << "Values in array1 by row are:" << endl;
   printArray( array1 );

   cout << "\nValues in array2 by row are:" << endl;
   printArray( array2 );

   cout << "\nValues in array3 by row are:" << endl;
   printArray( array3 );
} // end main
```

Fig. 6.18 | Initializing multidimensional arrays. (Part 1 of 3.)

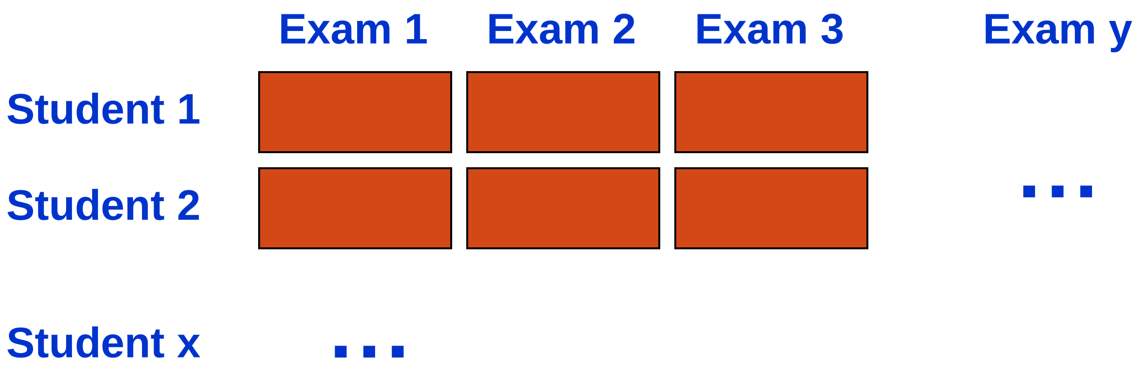
```cpp

// output array with two rows and three columns
void printArray( const int a[][ columns ] )
{
   // loop through array's rows
   for ( int i = 0; i < rows; i++ )
   {
      // loop through columns of current row
      for ( int j = 0; j < columns; j++ )
         cout << a[ i ][ j ] << ' ';

      cout << endl; // start new line of output
   } // end outer for
} // end function printArray
```

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

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

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

Fig. 6.18 | Initializing multidimensional arrays. (Part 3 of 3.)

6.9 Example

- Gradebook with 2-D array
  - Each student takes several exams
  - Display exam grades and average
  - Display grade distribution of all exam results
- 2-D array



Example: student grades in 2D array

- row: each student
- column: each exam

```
// Fig. 6.19: fig06_19.cpp
// Analyzing a two-dimensional array of grades.
#include <iostream>
#include <iomanip> // parameterized stream manipulators
using namespace std;

const int students = 10; // number of students
const int tests = 3; // number of tests

// function prototypes
int minimum( const int [][ tests ], int, int );
int maximum( const int [][ tests ], int, int );
double average( const int [], int );
void outputGrades( const int [][ tests ], int, int );
void outputBarChart( const int [][ tests ], int, int);

int main()
{
```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 1 of 9.)

```
   // two-dimensional array of student grades
   int studentGrades[ students ][ tests ] =
      { { 87, 96, 70 },
        { 68, 87, 90 },
        { 94, 100, 90 },
        { 100, 81, 82 },
        { 83, 65, 85 },
        { 78, 87, 65 },
        { 85, 75, 83 },
        { 91, 94, 100 },
        { 76, 72, 84 },
        { 87, 93, 73 } };

   // output the studentGrades array showing each student's average
   outputGrades( studentGrades, students, tests );

   // call functions minimum and maximum
   cout << "\nLowest of all the grades is "
      << minimum( studentGrades, students, tests )
      << "\nHighest of all the grades is "
      << maximum( studentGrades, students, tests ) << endl;

   // display a bar chart of the grades
   outputBarChart( studentGrades, students, tests );
} // end main
```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 2 of 9.)

```
// find the minimum of all the grades in the double array
int minimum( const int grades[][ tests ], int pupils, int exams )
{
   int lowGrade = 100; // assume lowest grade is 100

   // loop through rows of grades array
   for ( int student = 0; student < pupils; student++ )
   {
      // loop through columns of current row
      for ( int test = 0; test < exams; test++ )
      {
         // if current grade less than lowGrade, assign it to lowGrade
         if ( grades[ student ][ test ] < lowGrade )
            lowGrade = grades[ student ][ test ]; // new lowest grade
      } // end inner for
   } // end outer for

   return lowGrade; // return lowest grade
} // end function minimum
```

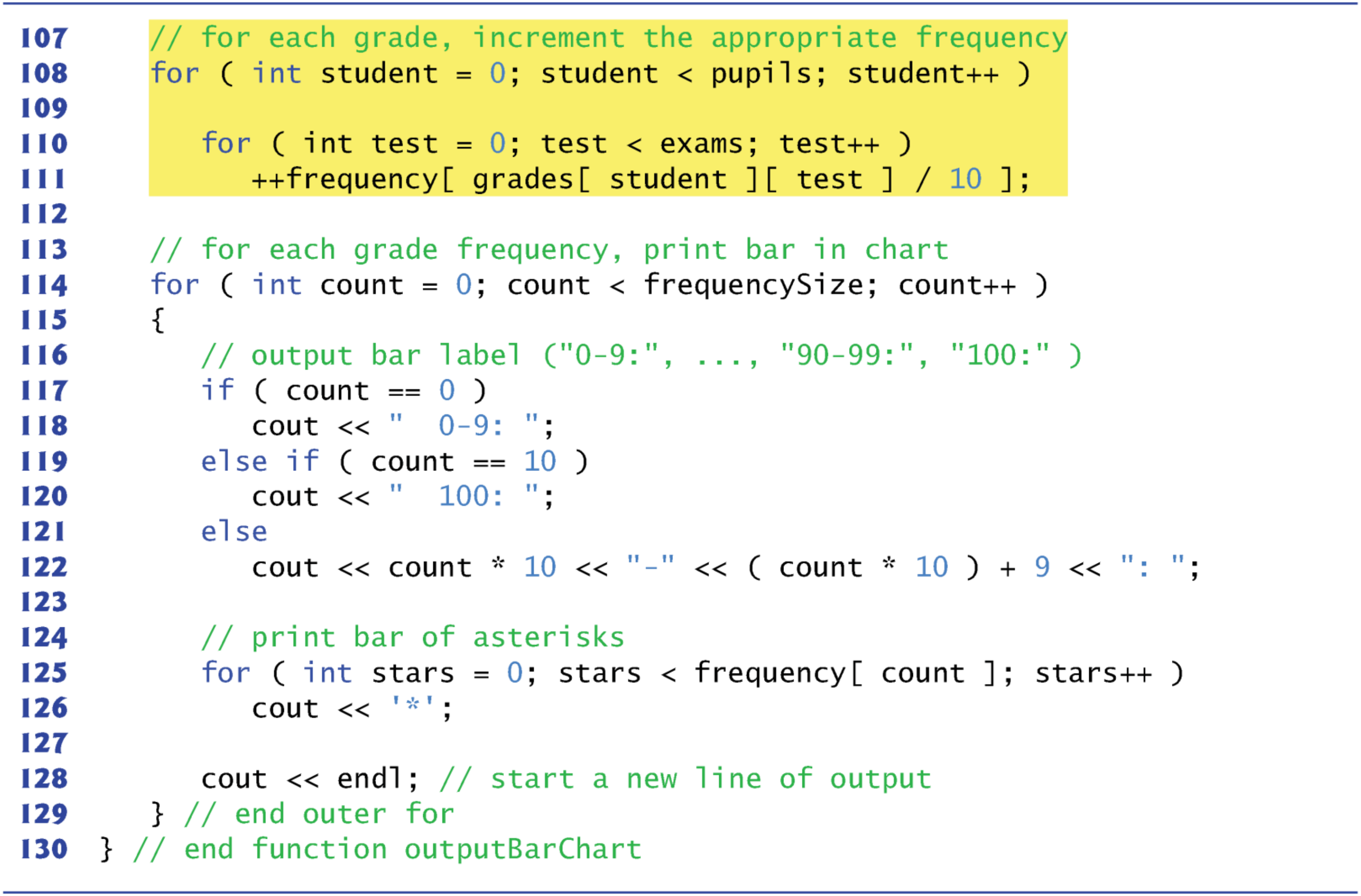
Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 3 of 9.)

```
// find the maximum of all the grades in the double array
int maximum( const int grades[][ tests ], int pupils, int exams )
{
   int highGrade = 0; // assume highest grade is 0

   // loop through rows of grades array
   for ( int student = 0; student < pupils; student++ )
   {
      // loop through columns of current row
      for ( int test = 0; test < exams; test++ )
      {
         // if current grade greater than highGrade, assign to highGrade
         if ( grades[ student ][ test ] > highGrade )
            highGrade = grades[ student ][ test ]; // new highest grade
      } // end inner for
   } // end outer for

   return highGrade; // return highest grade
} // end function maximum

```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 4 of 9.)

```
// determine average grade for particular set of grades
double average( const int setOfGrades[], const int gradeCount )
{
   int total = 0; // initialize total

   // sum grades in array
   for ( int grade = 0; grade < gradeCount; grade++ )
      total += setOfGrades[ grade ];

   // return average of grades
   return static_cast< double >( total ) / gradeCount;
} // end function average

// output bar chart displaying grade distribution
void outputBarChart( const int grades[][ tests ], int pupils, int exams )
{
   cout << "\nOverall grade distribution:" << endl;

   // stores frequency of grades in each range of 10 grades
   const int frequencySize = 11;
   int frequency[ frequencySize ] = {}; // initialize elements to 0

```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 5 of 9.)



```
   // for each grade, increment the appropriate frequency
   for ( int student = 0; student < pupils; student++ )

      for ( int test = 0; test < exams; test++ )
         ++frequency[ grades[ student ][ test ] / 10 ];

   // for each grade frequency, print bar in chart
   for ( int count = 0; count < frequencySize; count++ )
   {
      // output bar label ("0-9:", ..., "90-99:", "100:" )
      if ( count == 0 )
         cout << "  0-9: ";
      else if ( count == 10 )
         cout << "  100: ";
      else
         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";

      // print bar of asterisks
      for ( int stars = 0; stars < frequency[ count ]; stars++ )
         cout << '*';

      cout << endl; // start a new line of output
   } // end outer for
} // end function outputBarChart
```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 6 of 9.)

```

// output the contents of the grades array
void outputGrades( const int grades[][ tests ], int pupils, int exams )
{
   cout << "\nThe grades are:\n\n";
   cout << "            "; // align column heads

   // create a column heading for each of the tests
   for ( int test = 0; test < tests; test++ )
      cout << "Test " << test + 1 << "  ";

   cout << "Average" << endl; // student average column heading

   // create rows/columns of text representing array grades
   for ( int student = 0; student < pupils; student++ )
   {
      cout << "Student " << setw( 2 ) << student + 1;

      // output student's grades
      for ( int test = 0; test < exams; test++ )
         cout << setw( 8 ) << grades[ student ][ test ];

```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 7 of 9.)

```
      // call function average to calculate student's average;
      // pass row of grades and the value of tests as the arguments
      double averageGrade = average( grades[ student ], tests );
      cout << setw( 9 ) << setprecision( 2 ) << fixed
         << averageGrade << endl;
   } // end outer for
} // end function outputGrades
```

```
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
```

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 8 of 9.)