BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-1

Arrays

Instructor : PhD, Associate Professor Leyla Muradkhanli

Arrays

- An array is a group of memory locations related by the fact that they all have the same name and the same type.
- To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.
- Figure 6.1 shows an integer array called `c`.
- This array contains 12 elements.
- Any one of these elements may be referred to by giving the name of the array followed by the position number of the particular element in square brackets (`[]`).

Arrays (Cont.)

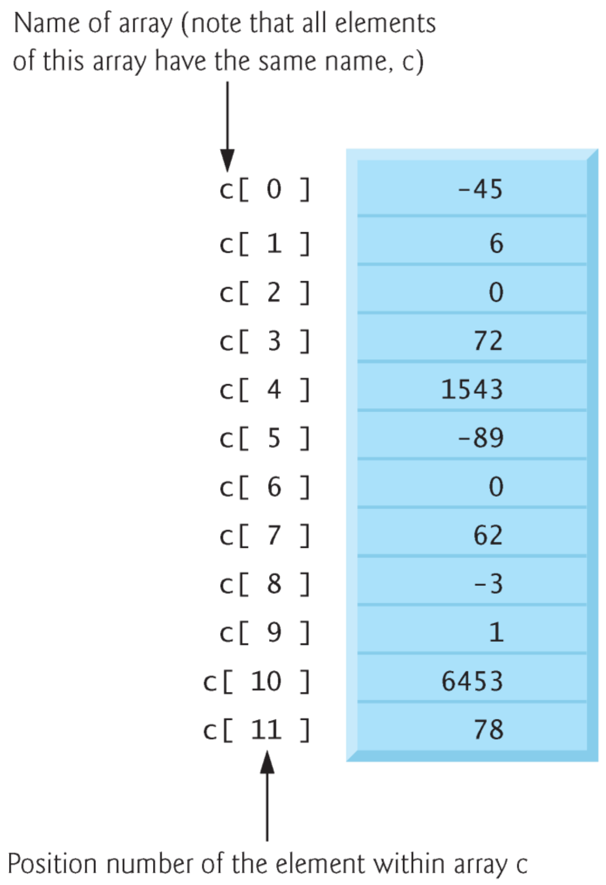
- The first element in every array is the zeroth element.
- Thus, the first element of array `c` is referred to as `c[0]`, the second element of array `c` is referred to as `c[1]`, the seventh element of array `c` is referred to as `c[6]`, and, in general, the *i*th element of array `c` is referred to as `c[i - 1]`.
- Array names, like other variable names, can contain only letters, digits and underscores.
- Array names cannot begin with a digit.
- The position number contained within square brackets is more formally called a subscript (or index).
- A subscript must be an integer or an integer expression.

Arrays (Cont.)

- If a program uses an expression as a subscript, then the expression is evaluated to determine the subscript.
- For example, if `a` = `5` and `b` = `6`, then the statement
    - `c[ a + b ] += 2;`
- adds 2 to array element `c[11]`.

Arrays (Cont.)

- Let’s examine array `c` (Fig. 6.1) more closely.
- The array’s name is `c`.
- Its 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, …, `c[11]`.
- The value stored in `c[0]` is `–45`, the value of `c[1]` is `6`, the value of `c[2]` is `0`, the value of `c[7]` is `62` and the value of `c[11]` is `78`.
- To print the sum of the values contained in the first three elements of array `c`, we’d write
  - `printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );`
- To divide the value of the seventh element of array `c` by `2` and assign the result to the variable `x`, we’d write



Name of array (note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

Fig. 6.1 | 12-element array.

Defining Arrays

- Arrays occupy space in memory.
- You specify the type of each element and the number of elements required by each array so that the computer may reserve the appropriate amount of memory.
- To tell the computer to reserve 12 elements for integer array `c`, use the definition
    - `int c[ 12 ];`

Defining Arrays (Cont.)

- The following definition
    - `int b[ 100 ], x[ 27 ];`

  reserves 100 elements for integer array `b` and 27 elements for integer array `x`.
- Arrays may contain other data types.

Array Examples

- Figure 6.3 uses `for` statements to initialize the elements of a 10-element integer array `n` to zeros and print the array in tabular format.
- The first `printf` statement (line 16) displays the column heads for the two columns printed in the subsequent `for` statement.

```c
/* Fig. 6.3: fig06_03.c
   initializing an array */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i; /* counter */

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = 0; /* set element at location i to 0 */
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array n in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 1 of 2.)

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 2 of 2.)

Array Examples (Cont.)

- The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, `{}`, containing a comma-separated list of initializers.
- Figure 6.4 initializes an integer array with 10 values (line 9) and prints the array in tabular format.

```c
/* Fig. 6.4: fig06_04.c
   Initializing an array with an initializer list */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array n */
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; /* counter */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 1 of 2.)

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 2 of 2.)

Array Examples (Cont.)

- If there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- For example, the elements of the array `n` in Fig. 6.3 could have been initialized to zero as follows:
  - `int n[ 10 ] = { 0 };`
- This explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array.

Array Examples (Cont.)

- It’s important to remember that arrays are not automatically initialized to zero.
- You must at least initialize the first element to zero for the remaining elements to be automatically zeroed.
- This method of initializing the array elements to `0` is performed at compile time for `static` arrays and at runtime for automatic arrays.

Array Examples (Cont.)

- The array definition
  - `int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };`

  causes a syntax error because there are six initializers and only five array elements.

Array Examples (Cont.)

- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list.
- For example,
    - `int n[] = { 1, 2, 3, 4, 5 };`

  would create a five-element array.

Array Examples (Cont.)

- Figure 6.5 initializes the elements of a 10-element array `s` to the values `2`, `4`, `6`, …, `20` and prints the array in tabular format.
- The values are generated by multiplying the loop counter by `2` and adding `2`.

```c
/* Fig. 6.5: fig06_05.c
   Initialize the elements of array s to the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10 /* maximum size of array */

/* function main begins program execution */
int main( void )
{
   /* symbolic constant SIZE can be used to specify array size */
   int s[ SIZE ]; /* array s has SIZE elements */
   int j; /* counter */

   for ( j = 0; j < SIZE; j++ ) { /* set the values */
      s[ j ] = 2 + 2 * j;
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array s in tabular format */
   for ( j = 0; j < SIZE; j++ ) {
      printf( "%7d%13d\n", j, s[ j ] );
   } /* end for */
```

Fig. 6.5 | Initialize the elements of array `s` to the even integers from 2 to 20. (Part 1 of 2.)

```c

   return 0; /* indicates successful termination */
} /* end main */
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

Fig. 6.5 | Initialize the elements of array `s` to the even integers from 2 to 20. (Part 2 of 2.)

Array Examples (Cont.)

- The `#define` preprocessor directive is introduced in this program.
- Line 4
    - `#define SIZE 10`

  defines a symbolic constant `SIZE` whose value is `10`.
- A symbolic constant is an identifier that is replaced with replacement text by the C preprocessor before the program is compiled.

Array Examples (Cont.)

- When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text `10`.
- Using symbolic constants to specify array sizes makes programs more scalable.
- In Fig. 6.5, we could have the first `for` loop (line 13) fill a 1000-element array by simply changing the value of `SIZE` in the `#define` directive from `10` to `1000`.
- If the symbolic constant `SIZE` had not been used, we'd have to change the program in three separate places to scale the program to handle 1000 array elements.

Array Examples (Cont.)

- If the `#define` preprocessor directive in line 4 is terminated with a semicolon, all occurrences of the symbolic constant `SIZE` in the program are replaced with the text `10;` by the preprocessor.
- This may lead to syntax errors at compile time, or logic errors at execution time.

Array Examples (Cont.)

- Figure 6.6 sums the values contained in the 12-element integer array `a`.
- The `for` statement’s body (line 16) does the totaling.

```c
/* Fig. 6.6: fig06_06.c
   Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array */
   int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
   int i; /* counter */
   int total = 0; /* sum of array */

   /* sum contents of array a */
   for ( i = 0; i < SIZE; i++ ) {
      total += a[ i ];
   } /* end for */

   printf( "Total of array element values is %d\n", total );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Total of array element values is 383
```

Fig. 6.6 | Computing the sum of the elements of an array.

Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets.
- For example, if array `hourlyTemperatures` has been defined as
    - `int hourlyTemperatures[ 24 ];`

  the function call
    - `modifyArray( hourlyTemperatures, 24 )`

  passes array `hourlyTemperatures` and its size to function `modifyArray`.

Passing Arrays to Functions (Cont.)

- Unlike `char` arrays that contain strings, other array types do not have a special terminator.
- For this reason, the size of an array is passed to the function, so that the function can process the proper number of elements.
- C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers’ original arrays.
- The name of the array evaluates to the address of the first element of the array.
- Because the starting address of the array is passed, the called function knows precisely where the array is stored.

Passing Arrays to Functions (Cont.)

- Figure 6.13 demonstrates the difference between passing an entire array and passing an array element.
- The program first prints the five elements of integer array `a` (lines 20–22).

Passing Arrays to Functions (Cont.)

- Next, `a` and its size are passed to function `modifyArray` (line 27), where each of `a`'s elements is multiplied by 2 (lines 54–55).
- Then `a` is reprinted in `main` (lines 32–34).
- As the output shows, the elements of `a` are indeed modified by `modifyArray`.
- Now the program prints the value of `a[3]` (line 38) and passes it to function `modifyElement` (line 40).
- Function `modifyElement` multiplies its argument by 2 (line 64) and prints the new value.
- When `a[3]` is reprinted in `main` (line 43), it has not been modified, because individual array elements are passed by value.

```c
/* Fig. 6.13: fig06_13.c
   Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

/* function prototypes */
void modifyArray( int b[], int size );
void modifyElement( int e );

/* function main begins program execution */
int main( void )
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
   int i; /* counter */

   printf( "Effects of passing entire array by reference:\n\nThe "
      "values of the original array are:\n" );

   /* output original array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[ i ] );
   } /* end for */

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 1 of 4.)

```
   printf( "\n" );

   /* pass array a to modifyArray by reference */
   modifyArray( a, SIZE );

   printf( "The values of the modified array are:\n" );

   /* output modified array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[ i ] );
   } /* end for */

   /* output value of a[ 3 ] */
   printf( "\n\n\nEffects of passing array element "
      "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );

   modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */

   /* output value of a[ 3 ] */
   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
   return 0; /* indicates successful termination */
} /* end main */

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 4.)

```
/* in function modifyArray, "b" points to the original array "a"
   in memory */
void modifyArray( int b[], int size )
{
   int j; /* counter */

   /* multiply each array element by 2 */
   for ( j = 0; j < size; j++ ) {
      b[ j ] *= 2;
   } /* end for */
} /* end function modifyArray */

/* in function modifyElement, "e" is a local copy of array element
   a[ 3 ] passed from main */
void modifyElement( int e )
{
   /* multiply parameter by 2 */
   printf( "Value in modifyElement is %d\n", e *= 2 );
} /* end function modifyElement */
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 3 of 4.)

```
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:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[ 3 ] is 6
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 4 of 4.)

Passing Arrays to Functions (Cont.)

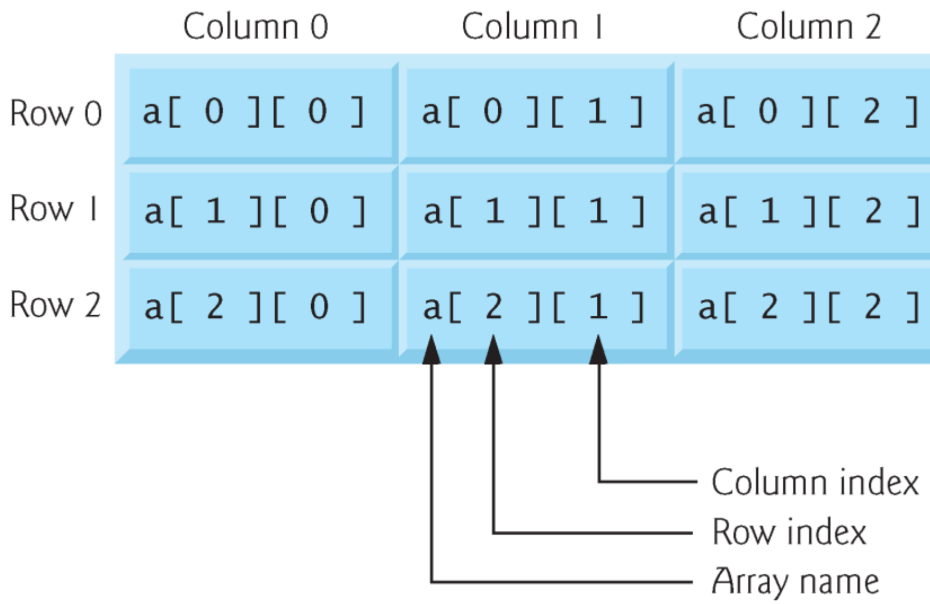
- There may be situations in your programs in which a function should not be allowed to modify array elements.
- Because arrays are always passed by reference, modification of values in an array is difficult to control.
- C provides the type qualifier `const` to prevent modification of array values in a function.
- When an array parameter is preceded by the `const` qualifier, the array elements become constant in the function body, and any attempt to modify an element of the array in the function body results in a compile-time error.
- This enables you to correct a program so it does not attempt to modify array elements.

Multiple-Subscripted Arrays

- Arrays in C can have multiple subscripts.
- A common use of multiple-subscripted arrays (also called multidimensional arrays) is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we must specify two subscripts: The first (by convention) identifies the element’s row and the second (by convention) identifies the element’s column.
- Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays.

Multiple-Subscripted Arrays (Cont.)

- Multiple-subscripted arrays can have more than two subscripts.
- Figure 6.20 illustrates a double-subscripted 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*



	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 ]`

Column index

Row index

Array name

Fig. 6.20 | Double-subscripted array with three rows and four columns.

Multiple-Subscripted Arrays (Cont.)

- Every element in array `a` is identified in Fig. 6.20 by an element name of the form `a[i][j]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`.
- The names of the elements in the first row all have a first subscript of `0`; the names of the elements in the fourth column all have a second subscript of `3`.

Multiple-Subscripted Arrays (Cont.)

- A multiple-subscripted array can be initialized when it’s defined, much like a single-subscripted array.
- For example, a double-subscripted array `int b[2][2]` could be defined and initialized with
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
- The values are grouped by row in braces.
- The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1.
- So, the values `1` and `2` initialize elements `b[0][0]` and `b[0][1]`, respectively, and the values `3` and `4` initialize elements `b[1][0]` and `b[1][1]`, respectively.

Multiple-Subscripted Arrays (Cont.)

- If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.
- Thus,
  - `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

  would initialize b[0][0] to 1, b[0][1] to 0, b[1][0] to 3 and b[1][1] to 4.

Multiple-Subscripted Arrays (Cont.)

- Figure 6.21 demonstrates defining and initializing double-subscripted arrays.
- The program defines three arrays of two rows and three columns (six elements each).
- The definition of `array1` (line 11) provides six initializers in two sublists.
- The first sublist initializes the first row (i.e., row 0) of the array to the values 1, 2 and 3; and the second sublist initializes the second row (i.e., row 1) of the array to the values 4, 5 and 6.

```c
/* Fig. 6.21: fig06_21.c
   Initializing multidimensional arrays */
#include <stdio.h>

void printArray( const int a[][ 3 ] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   /* initialize array1, array2, array3 */
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   printf( "Values in array1 by row are:\n" );
   printArray( array1 );

   printf( "Values in array2 by row are:\n" );
   printArray( array2 );

   printf( "Values in array3 by row are:\n" );
   printArray( array3 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 3.)

```c

/* function to output array with two rows and three columns */
void printArray( const int a[][ 3 ] )
{
   int i; /* row counter */
   int j; /* column counter */

   /* loop through rows */
   for ( i = 0; i <= 1; i++ ) {

      /* output column values */
      for ( j = 0; j <= 2; j++ ) {
         printf( "%d ", a[ i ][ j ] );
      } /* end inner for */

      printf( "\n" ); /* start new line of output */
   } /* end outer for */
} /* end function printArray */
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 2 of 3.)

```
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. 6.21 | Initializing multidimensional arrays. (Part 3 of 3.)

Multiple-Subscripted Arrays (Cont.)

- If the braces around each sublist are removed from the `array1` initializer list, the compiler initializes the elements of the first row followed by the elements of the second row.
- The definition of `array2` (line 12) provides five initializers.
- The initializers are assigned to the first row, then the second row.
- Any elements that do not have an explicit initializer are initialized to zero automatically, so `array2[1][2]` is initialized to 0.
- The definition of `array3` (line 13) provides three initializers in two sublists.

Multiple-Subscripted Arrays (Cont.)

- The sublist for the first row explicitly initializes the first two elements of the first row to 1 and 2.
- The third element is initialized to zero.
- The sublist for the second row explicitly initializes the first element to 4.
- The last two elements are initialized to zero.
- The program calls `printArray` (lines 27–43) to output each array’s elements.
- The function definition specifies the array parameter as `const int a[][3]`.
- When we receive a single-subscripted array as a parameter, the array brackets are empty in the function’s parameter list.

Multiple-Subscripted Arrays (Cont.)

- In a double-subscripted array, each row is basically a single-subscripted array.
- To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array.
- Thus, when accessing `a[1][2]` in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1).
- Then, the compiler accesses the third element of that row (element 2).

Multiple-Subscripted Arrays (Cont.)

- Many common array manipulations use `for` repetition statements.
- For example, the following statement sets all the elements in the third row of array `a` in Fig. 6.20 to zero:

  -
    ```
    for ( column = 0; column <= 3; column++ ) {
       a[ 2 ][ column ] = 0;
    }
    ```
- We specified the third row, therefore we know that the first subscript is always `2` (again, `0` is the first row and `1` is the second).

Multiple-Subscripted Arrays (Cont.)

- The `loop` varies only the second subscript (i.e., the column).
- The preceding `for` statement is equivalent to the assignment statements:

  -
    ```
    a[ 2 ][ 0 ] = 0;
    a[ 2 ][ 1 ] = 0;
    a[ 2 ][ 2 ] = 0;
    a[ 2 ][ 3 ] = 0;
    ```

Multiple-Subscripted Arrays (Cont.)

- The following nested for statement determines the total of all the elements in array a.

  -
    ```
    total = 0;
    ```

  -
    ```
    for ( row = 0; row <= 2; row++ ) {
       for ( column = 0; column <= 3; column++ ) {
          total += a[ row ][ column ];
       }
    }
    ```
- The for statement totals the elements of the array one row at a time.

Multiple-Subscripted Arrays (Cont.)

- The outer `for` statement begins by setting `row` (i.e., the row subscript) to `0` so that the elements of the first row may be totaled by the inner `for` statement.
- The outer `for` statement then increments `row` to `1`, so the elements of the second row can be totaled.
- Then, the outer `for` statement increments `row` to `2`, so the elements of the third row can be totaled.
- The result is printed when the nested `for` statement terminates.