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

• 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
### Fig. 6.20  |  Double-subscripted array with three rows and four columns.
Multiple-Subscripted Arrays

• 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.
Common Programming Error 6.9
Referencing a double-subscripted array element as `a[x, y]` instead of `a[x][y]`. C interprets `a[x][y]` as `a[y][x]`, and as such it does not cause a compilation error.
Multiple-Subscripted Arrays

- 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

• If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.

• Thus,

```c
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

• 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: f1g06_21.c */
#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.)
// 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 %d %d
", a[ i ][ j ] );
        } /* end inner for */

    } /* end outer for */

    printf( "\n" ); /* start new line of output */
}
/* 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. 6.2i  Initializing multidimensional arrays. (Part 3 of 3.)
• 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

- 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.
The first subscript of a multiple-subscripted array is not required either, but all subsequent subscripts are required.

The compiler uses these subscripts to determine the locations in memory of elements in multiple-subscripted arrays.

All array elements are stored consecutively in memory regardless of the number of subscripts.

In a double-subscripted array, the first row is stored in memory followed by the second row.

Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an element in the array.
Multiple-Subscripted Arrays

- 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).
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:

```cpp
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).
• 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

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

```c
    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

• 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.
Multiple-Subscripted Arrays

• Figure 6.22 performs several other common array manipulations on 3-by-4 array `studentGrades` using `for` statements.

• Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester.

• The array manipulations are performed by four functions.

• Function `minimum` (lines 43–62) determines the lowest grade of any student for the semester.
Multiple-Subscripted Arrays

- Function `maximum` (lines 65–84) determines the highest grade of any student for the semester.
- Function `average` (lines 87–98) determines a particular student’s semester average.
- Function `printArray` (lines 101–120) outputs the double-subscripted array in a neat, tabular format.
/* Fig. 6.22: fig06_22.c */
Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

/* function prototypes */
int minimum( const int grades[][ EXAMS ], int pupils, int tests );
int maximum( const int grades[][ EXAMS ], int pupils, int tests );
double average( const int setOfGrades[], int tests );
void printArray( const int grades[][ EXAMS ], int pupils, int tests );

/* function main begins program execution */
int main( void )
{
    int student; /* student counter */

    /* initialize student grades for three students (rows) */
    const int studentGrades[ STUDENTS ][ EXAMS ] =
    {
        { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 };
    }

Fig. 6.22 | Double-subscripted arrays example. (Part 1 of 7.)
/* output array studentGrades */
printf( "The array is:\n" );
printArray( studentGrades, STUDENTS, EXAMS );

/* determine smallest and largest grade values */
printf( "\n\nLowest grade: %d\nHighest grade: %d\n", 
   minimum( studentGrades, STUDENTS, EXAMS ),
   maximum( studentGrades, STUDENTS, EXAMS ) );

/* calculate average grade for each student */
for ( student = 0; student < STUDENTS; student++ ) {
   printf( "The average grade for student %d is %.2f\n", 
      student, average( studentGrades[ student ], EXAMS ) );
} /* end for */

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

Fig. 6.22 | Double-subscripted arrays example. (Part 2 of 7.)
/* Find the minimum grade */
int minimum( const int grades[][ EXAMS ], int pupils, int tests )
{
    int i; /* student counter */
    int j; /* exam counter */
    int lowGrade = 100; /* initialize to highest possible grade */

    /* loop through rows of grades */
    for ( i = 0; i < pupils; i++ ) {

        /* loop through columns of grades */
        for ( j = 0; j < tests; j++ ) {

            if ( grades[ i ][ j ] < lowGrade ) {
                lowGrade = grades[ i ][ j ];
        } /* end if */
    } /* end inner for */
} /* end outer for */

return lowGrade; /* return minimum grade */
} /* end function minimum */

Fig. 6.22  | Double-subscripted arrays example. (Part 3 of 7.)
/* Find the maximum grade */
int maximum( const int grades[][ EXAMS ], int pupils, int tests )
{
    int i; /* student counter */
    int j; /* exam counter */
    int highGrade = 0; /* initialize to lowest possible grade */

    /* loop through rows of grades */
    for ( i = 0; i < pupils; i++ ) {

        /* loop through columns of grades */
        for ( j = 0; j < tests; j++ ) {

            if ( grades[ i ][ j ] > highGrade ) {
                highGrade = grades[ i ][ j ];
            } /* end if */
        } /* end inner for */
    } /* end outer for */

    return highGrade; /* return maximum grade */
} /* end function maximum */
/* Determine the average grade for a particular student */
double average( const int setOfGrades[], int tests )
{
    int i; /* exam counter */
    int total = 0; /* sum of test grades */

    /* total all grades for one student */
    for ( i = 0; i < tests; i++ ){
        total += setOfGrades[ i ];
    } /* end for */

    return ( double ) total / tests; /* average */
} /* end function average */

**Fig. 6.22**  |  Double-subscripted arrays example. (Part 5 of 7.)
/* Print the array */
void printArray( const int grades[][EXAMS], int pupils, int tests )
{
    int i; /* student counter */
    int j; /* exam counter */

    /* output column heads */
    printf( " [0] [1] [2] [3]" );

    /* output grades in tabular format */
    for ( i = 0; i < pupils; i++ ) {
        /* output label for row */
        printf( "\nStudentGrades[%d] ", i );

        /* output grades for one student */
        for ( j = 0; j < tests; j++ ) {
            printf( "%-5d", grades[i][j] );
        } /* end inner for */
    } /* end outer for */
} /* end function printArray */

Fig. 6.22  | Double-subscripted arrays example. (Part 6 of 7.)
The array is:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\hline
\text{studentGrades}[0] & 77 & 68 & 86 & 73 \\
\text{studentGrades}[1] & 96 & 87 & 89 & 78 \\
\text{studentGrades}[2] & 70 & 90 & 86 & 81 \\
\end{array}
\]

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75

Fig. 6.22  |  Double-subscripted arrays example. (Part 7 of 7.)
Functions `minimum`, `maximum` and `printArray` each receive three arguments—the `studentGrades` array (called `grades` in each function), the number of students (rows of the array) and the number of exams (columns of the array).

Each function loops through array `grades` using nested `for` statements.
The following nested for statement is from the function `minimum` definition:

```c
/* loop through rows of grades */
for ( i = 0; i < pupils; i++ ) {
    /* loop through columns of grades */
    for ( j = 0; j < tests; j++ ) {
        if ( grades[ i ][ j ] < lowGrade ) {
            lowGrade = grades[ i ][ j ];
        }
    }
}
```
Multiple-Subscripted Arrays

• The outer `for` statement begins by setting `i` (i.e., the row subscript) to 0 so that the elements of the first row (i.e., the grades of the first student) can be compared to variable `lowGrade` in the body of the inner `for` statement.

• The inner `for` statement loops through the four grades of a particular row and compares each grade to `lowGrade`.

• If a grade is less than `lowGrade`, `lowGrade` is set to that grade.

• The outer `for` statement then increments the row subscript to 1.

• The elements of the second row are compared to variable `lowGrade`. 
Multiple-Subscripted Arrays

- The outer \texttt{for} statement then increments the row subscript to 2.
- The elements of the third row are compared to variable \texttt{lowGrade}.
- When execution of the nested structure is complete, \texttt{lowGrade} contains the smallest grade in the double-subscripted array.
- Function \texttt{maximum} works similarly to function \texttt{minimum}.
- Function \texttt{average} (line 87) takes two arguments—a single-subscripted array of test results for a particular student called \texttt{setOfGrades} and the number of test results in the array.
Multiple-Subscripted Arrays

- When `average` is called, the first argument `studentGrades[student]` is passed.
- This causes the address of one row of the double-subscripted array to be passed to `average`.
- The argument `studentGrades[1]` is the starting address of the second row of the array.
- Remember that a double-subscripted array is basically an array of single-subscripted arrays and that the name of a single-subscripted array is the address of the array in memory.
- Function `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.