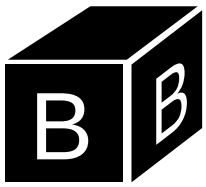

Chapter *18*

Two-Dimensional Arrays

Sometimes you need to store information not as a single list of values, but as a table of values. Tables have rows and columns. The Component Pascal data structure that corresponds to a table is a two-dimensional array. In the same way that vector is another name for a one-dimensional array, matrix is another name for a two-dimensional array.

Matrix input/output

Figure 18.1 shows a tool dialog box for inputting a two-dimensional array of real values from the focus window and outputting the array to a new window. The dialog box is opened as a tool, so the scanner will be attached to the model of the view that was previously the focus window, which is titled "untitled 1" in the figure. When the user clicks the Load Matrix button, the program scans the real numbers into a two-dimensional array. When the user clicks the Display Matrix button a new window appears, which is titled "untitled 2" in the figure, that contains the values previously scanned into the two-dimensional array.

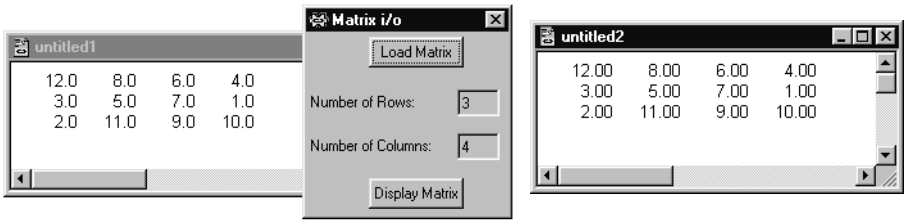

Figure 18.1
Input and output for a two-dimensional array of real values.

Figure 18.2 is the implementation of the dialog box of Figure 18.1. It defines global variable matrix to be a two-dimensional array with the declaration

```
matrix: ARRAY 32, 32 OF REAL
```

The two indices, 32 and 32, is what makes the array two-dimensional. The first index numbers the rows and the second index numbers the columns. matrix contains storage for $32 \times 32 = 1024$ values in 32 rows and 32 columns, although the program does not in general use all the storage.

```
MODULE Pbox18A;
   IMPORT Dialog, TextModels, TextViews, Views, TextControllers, PboxMappers;
   TYPE
      Matrix = ARRAY 32, 32 OF REAL;
   VAR
      d*: RECORD
         numRows-, numCols-: INTEGER;
      END;
      matrix: Matrix;

   PROCEDURE LoadMatrix*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanRealMatrix(matrix, d.numRows, d.numCols);
         Dialog.Update(d)
      END;
   END LoadMatrix;

   PROCEDURE DisplayMatrix*;
      VAR
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      md := TextModels.dir.New();
      fm.ConnectTo(md);
      fm.WriteRealMatrix(matrix, d.numRows, d.numCols, 8, 2);
      vw := TextViews.dir.New(md);
      Views.OpenView(vw)
   END DisplayMatrix;

BEGIN
   d.numRows := 0; d.numCols := 0;
END Pbox18A.
```

Figure 18.2
A procedure that inputs a matrix and outputs it to a new window.

Figure 18.3 shows how the cells in the matrix are numbered. matrix[2, 3] is the component in row two, column three. In general, matrix[i, j] is the component in row i, column j.

Procedure LoadMatrix in Figure 18.2 contains our usual MVC suspects for connecting to the focus window. It inputs the values by calling the imported procedure ScanRealMatrix from module PboxMappers. Here is the documentaion for ScanRealMatrix.

matrix[0,0] 12.0	matrix[0, 1] 8.0	matrix[0, 2] 6.0	matrix[0, 3] 4.0
matrix[1, 0] 3.0	matrix[1, 1] 5.0	matrix[1, 2] 7.0	matrix[1, 3] 1.0
matrix[2, 0] 2.0	matrix[2, 1] 11.0	matrix[2, 2] 9.0	matrix[2, 3] 10.0

Figure 18.3
Indices for a two-dimensional array of real values.

```
PROCEDURE (VAR s: Scanner) ScanRealMatrix (OUT mat: ARRAY OF ARRAY OF REAL;
        OUT numR, numC: INTEGER), NEW
Pre
s is connected to a text model.    20
Sequences of characters scanned represent in-range real or integer values.    21
All nonempty rows have the same number of values.    22
Number of rows in text model <= LEN(mat, 0).    Index out of range.
Number of columns in text model <= LEN(mat, 1).    Index out of range.
Post
mat gets all the values scanned up to the end of the text model to which s is connected.
numR gets the number of rows scanned.
numC gets the number of columns scanned.
The values are stored  at v[0..numR - 1, 0..numC - 1].
```

The operation of ScanRealMatrix

ScanRealMatrix expects each row of the matrix to be on a separate line. It scans all the values on the first line into the first row of the two-dimensional array mat, counting how many values are on the first line of the text model. It then scans each of the other lines in turn, counting the number of values on each line. If it detects a different number of values from the number on the first line, it terminates with a trap. ScanRealMatrix allows you to have any number of leading or trailing blank lines, which it ignores in the scan. You can even have blank lines between two lines of numbers and the embedded blank line will be ignored as well. At the completion of the scan, numR will contain the number of rows in mat, and numC will contain the number of columns.

LEN with a two-dimensional array

The Component Pascal function LEN returns the number of elements in a one-dimensional array. You can also use LEN in an array with more than one dimension, but it requires two parameters instead of just one. The function call LEN(arr, n) returns the number of cells in the nth dimension of array arr starting with n = 0 for the first dimension. For a two-dimensional array, LEN(mat, 0) is the number of rows of mat, and LEN(mat, 1) is the number of cells in each row. A precondition for procedure ScanRealMatrix is that matrix mat has enough rows and columns to store the values that are in the model to which s is connected.

Procedure DisplayMatrix outputs the values by calling WriteRealMatrix, also from PboxMappers. Here is the documentaion for WriteRealMatrix.

PROCEDURE (VAR f: Formatter) **WriteRealMatrix** (IN mat: ARRAY OF ARRAY OF REAL;
numR, numC, minWidth, dec: INTEGER), NEW
Pre
f is connected to a text model. 20
numR <= LEN(mat, 0). Index out of range.
numC <= LEN(mat, 1). Index out of range.
Post
The first numR rows and numC columns of mat are written to the text model to which f
is connected, each with a field width of minWidth and dec places past the decimal point.
If minWidth is too small to contain a value of mat it expands to accommodate the value.

The operation of WriteRealMatrix

Parameter mat is called by constant reference, because its values are defined when the procedure is called and its values are not to be changed. To write the matrix you must supply the number of rows and columns in numR and numC. You also supply a value for minWidth, the field width for each real value, and for dec, the number of places you want to display past the decimal point.

Printing a column

Figure 18.4 shows the input and output of a program that prints a column of a matrix to a new window. The dialog box allows the user to load the values from the focus window into the matrix as in Figure 18.2. It also contains a text field for the user to enter the column to display.

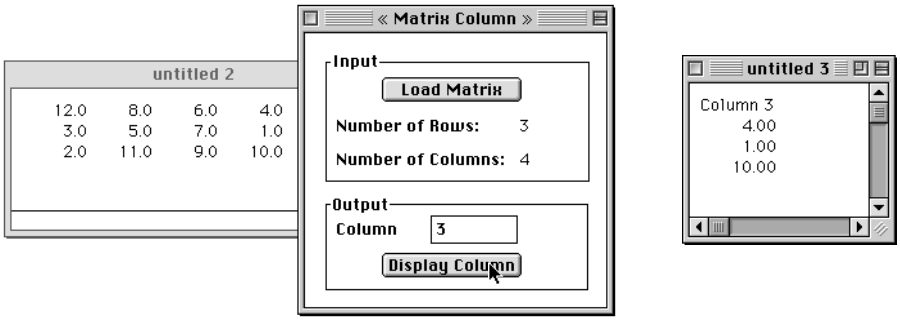

Figure 18.4
Printing a column of a matrix to a new window.

Before looking at the program that implements the dialog box of Figure 18.4, try to solve the problem yourself. Suppose you are given a two-dimensional array matrix with d.numRows rows, and an integer variable, d.column. The problem is to output all the values from matrix in that column. For example, if d.column has the value of 3, the code should output

```
4.00
1.00
10.00
```

which are all the components in column three. The specific output statements with

formatter fm are

```
fm.WriteReal(matrix[0, 3], 8, 2); fm.WriteLn
fm.WriteReal(matrix[1, 3], 8, 2); fm.WriteLn
fm.WriteReal(matrix[2, 3], 8, 2); fm.WriteLn
```

assuming a field width of 8 and two places past the decimal point. In a FOR loop the statements are

```
FOR i := 0 TO 2 DO
   fm.WriteReal(matrix[i, 3], 8, 2); fm.WriteLn
END
```

for an array with values in three rows. Because the value of numRows specifies how many rows of data are in the array, you must replace the 2 with d.numRows - 1. For a general column, you must replace the 3 with d.column. Here is the code for printing the column.

```
FOR i := 0 TO d.numRows - 1 DO
   fm.WriteReal(matrix[i, d.column], 8, 2); fm.WriteLn
END
```

Figure 18.5 shows the module that implements the dialog box of Figure 18.4.

```
MODULE Pbox18B;
   IMPORT Dialog, TextModels, TextViews, Views, TextControllers, PboxMappers;
   TYPE
      Matrix = ARRAY 32, 32 OF REAL;
   VAR
      d*: RECORD
         numRows-, numCols-: INTEGER;
         column*: INTEGER;
      END;
      matrix: Matrix;

   PROCEDURE LoadMatrix*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanRealMatrix(matrix, d.numRows, d.numCols);
         Dialog.Update(d)
      END;
   END LoadMatrix;
```

Figure 18.5
A program to print a column of a matrix. The dialog is activated from a menu selection.

```
    PROCEDURE DisplayColumn*;
        VAR
            md: TextModels.Model;
            vw: TextViews.View;
            fm: PboxMappers.Formatter;
            i: INTEGER;
    BEGIN
        md := TextModels.dir.New();
        fm.ConnectTo(md);
        IF (0 <= d.column) & (d.column < d.numCols) THEN
            fm.WriteString("Column "); fm.WriteInt(d.column, 1); fm.WriteLn;
            FOR i := 0 TO d.numRows - 1 DO
                fm.WriteReal(matrix[i, d.column], 8, 2); fm.WriteLn
            END
        ELSE
            fm.WriteString("That column is not in the array")
        END;
        vw := TextViews.dir.New(md);
        Views.OpenView(vw)
    END DisplayColumn;

BEGIN
    d.numRows := 0; d.numCols := 0; d.column := 0;
END Pbox18B.
```

Finding the largest in a row

The next illustration of a two-dimensional array involves finding the largest element in a row of a matrix. The problem is to complete the statements for the function procedure

```
PROCEDURE MaxInRow (IN mat: ARRAY OF ARRAY OF REAL; row, numCols: INTEGER): REAL;
```

Because mat is called by constant reference, assume its values are defined when the procedure is called. Because row and numCols are called by value, their initial values are also defined. numCols is the number of columns in mat, and row is the row from which we want the maximum value. For example, if the values in mat are again

```
 12.0   8.0   6.0   4.0
  3.0   5.0   7.0   1.0
  2.0  11.0   9.0  10.0
```

and if row has the value 2, the function should return the value 11.0. A precondition is that there is at least one row and one column. Otherwise there can be no maximum.

Assume you have a local variable max to store the maximum value found. The specific statements to compute max for row 2 are

```
max := mat[2, 0];
```

```
IF mat[2, 1] > max THEN
   max := mat[2, 1]
END;
IF mat[2, 2] > max THEN
   max := mat[2, 2]
END;
IF mat[2, 3] > max THEN
   max := mat[2, 3]
END
```

After execution, max has the maximum value of 11.0 from row two (the third row). The corresponding FOR statement assuming local integer variable j is

```
max := mat[2, 0];
FOR j := 1 TO 3 DO
   IF mat[2, j] > max THEN
      max := mat[2, j]
   END
END;
```

for an array with values in four columns. For numCols in general, j in the FOR loop ranges from 1 to numCols - 1. And for a general row, the FOR loop is shown in Figure 18.6.

```
PROCEDURE MaxInRow (IN mat: ARRAY OF ARRAY OF REAL;
      row, numCols: INTEGER): REAL;
VAR
   max: REAL;
   j: INTEGER;
BEGIN
   ASSERT((0 <= row) & (row < LEN(mat, 0)), 20);
   ASSERT((0 < numCols) & (numCols <= LEN(mat, 1)), 21);
   max := mat[row, 0];
   FOR j := 1 TO numCols - 1 DO
      IF mat[row, j] > max THEN
         max := mat[row, j]
      END
   END;
   RETURN max
END MaxInRow;
```

Figure 18.6
A function procedure that returns the maximum value in a row of a matrix.

Matrix multiplication

The next illustration deals with multiplication of two matrices, a and b, with the product in c. To multiply matrix a times matrix b, the number of columns of a must equal the number of rows of b. The integer variable numRa is the number of rows in matrix a, numCaRb is the number of columns in matrix a and rows in matrix b, and numCb is the number of columns in b. The product c will have the same number of

rows as a and the same number of columns as b.

For example, if a and b have the values

a				b		
1.0	3.0	–1.0	2.0	–1.0	2.0	5.0
0.0	4.0	1.0	–2.0	–3.0	0.0	4.0
				1.0	–2.0	3.0
				3.0	2.0	–4.0

then the product c should have the values

c		
–5.00	8.00	6.00
–17.00	–6.00	27.00

With these matrices, numRa is 2, numCaRb is 4, and numCb is 3.

Each value of c comes from multiplying a row of a with a column of b. For example, the element in the second row and third column of c comes from multiplying the second row of a with the third column of b. You multiply a row with a column by multiplying corresponding components from left to right in the row and from top to bottom in the column. Then you add the products. This specific case is

```
0.0 * 5.0 + 4.0 * 4.0 + 1.0 * 3.0 + (-2.0) * (-4.0)
```

which is 27.0. In general, the element in row i and column j of c comes from multiplying row i of a with column j of b.

Our problem is to write the proper procedure

```
PROCEDURE Multiply (IN a, b: ARRAY OF ARRAY OF REAL;
   numRa, numCaRb, numCb: INTEGER; OUT c: ARRAY OF ARRAY OF REAL);
```

Because a and b are called by constant reference, their initial values are defined. The problem is to compute new values for the matrix product c, which is called by result.

Start with the specific case above. For row one and column two of c, you need to compute

```
a[1, 0] * b[0, 2] + a[1, 1] * b[1, 2] + a[1, 2] * b[2, 2] + a[1, 3] * b[3, 2]
```

In a FOR loop, that is

```
sum := 0.0;
FOR k := 0 TO 3 DO
   sum := sum + a [1, k] * b [k, 2]
END;
c [1, 2] := sum
```

where sum is a real variable.

This code is for arrays with four columns in a and four rows in b. For the more

general case of numCaRb columns and rows, k in the FOR statement ranges from 0 to numCaRb - 1. Also, this computation is for row one, column two of c. In the more general case of row i and column j of c the code is

```
sum := 0.0;
FOR k := 0 TO numCaRb - 1 DO
   sum := sum + a[i, k] * b[k, j]
END;
c[i, j] := sum
```

This computation must be done for every row and column of c. So, it must be nested in a nested loop with i ranging from 0 to numRa - 1 and j ranging from 0 to numCb - 1. The final code for matrix multiplication is in procedure Multiply in Figure 18.8. Figure 18.7 shows the corresponding dialog box.

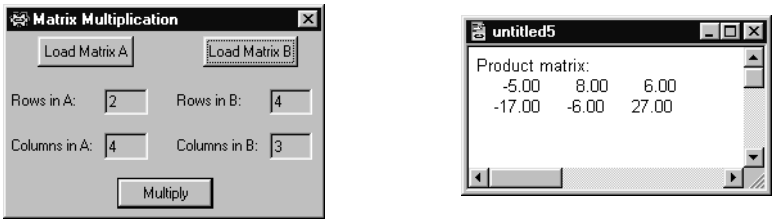

Figure 18.7
Output for the product of two matrices.

```
MODULE Pbox18C;
   IMPORT Dialog,  TextModels, TextViews, Views, TextControllers, PboxMappers;
   VAR
      d*: RECORD
         numRowA-, numColA-, numRowB-, numColB-: INTEGER;
      END;
      matrixA, matrixB: ARRAY 32, 32 OF REAL;

   PROCEDURE LoadA*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanRealMatrix(matrixA, d.numRowA, d.numColA);
         Dialog.Update(d)
      END
   END LoadA;
```

Figure 18.8
A program that multiplies two matrices.

Figure 18.8
Continued.

```
PROCEDURE LoadB*;
    VAR
        md: TextModels.Model;
        cn: TextControllers.Controller;
        sc: PboxMappers.Scanner;
BEGIN
    cn := TextControllers.Focus();
    IF cn # NIL THEN
        md := cn.text;
        sc.ConnectTo(md);
        sc.ScanRealMatrix(matrixB, d.numRowB, d.numColB);
        Dialog.Update(d)
    END
END LoadB;

PROCEDURE Multiply (IN a, b: ARRAY OF ARRAY OF REAL;
        numRa, numCaRb, numCb: INTEGER;
        OUT c: ARRAY OF ARRAY OF REAL);
VAR
    sum: REAL;
    i, j, k: INTEGER;
BEGIN
    ASSERT((0 <= numRa) & (numRa <= LEN(a, 0)) & (numRa <= LEN(c, 0))
        & (0 <= numCaRb) & (numCaRb <= LEN(a, 1)) & (numCaRb <= LEN(b, 0))
        & (0 <= numCb) & (numCb <= LEN(b, 1)) & (numCb <= LEN(c, 1)), 20);
    FOR i := 0 TO numRa - 1 DO
        FOR j := 0 TO numCb - 1 DO
            sum := 0.0;
            FOR k := 0 TO numCaRb - 1 DO
                sum := sum + a[i, k] * b[k, j]
            END;
            c[i, j] := sum
        END
    END
END Multiply;
```

Figure 18.8
Continued.

```
   PROCEDURE Compute*;
      VAR
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
         matrixC: ARRAY 32, 32 OF REAL;
   BEGIN
      md := TextModels.dir.New();
      fm.ConnectTo(md);
      IF d.numColA = d.numRowB THEN
         Multiply(matrixA, matrixB, d.numRowA, d.numColA, d.numColB, matrixC);
         fm.WriteString("Product matrix:"); fm.WriteLn;
         fm.WriteRealMatrix(matrixC, d.numRowA, d.numColB, 8, 2)
      ELSE
         fm.WriteString("The number of columns in A must equal the number of rows in B.")
      END;
      vw := TextViews.dir.New(md);
      Views.OpenView(vw)
   END Compute;

BEGIN
   d.numRowA := 0; d.numColA := 0; d.numRowB := 0; d.numColB := 0;
END Pbox18C.
```

To determine the statement execution count of procedure Multiply number the executable statements as follows.

The executable statements of procedure Multiply

Statement number	*Executable statement*
(1)	FOR i := 0 TO numRa - 1 DO
(2)	FOR j := 0 TO numCb - 1 DO
(3)	sum := 0.0;
(4)	FOR k := 0 TO numCaRb - 1 DO
(5)	sum := sum + a[i, k] * b[k, j]
(6)	c[i, j] := sum

For simplicity, assume that both matrix a and b have *n* rows and *n* columns. Then, the test for each for loop executes $n + 1$ times and each statement in the body of the for loop executes *n* times. Figure 18.9 shows the statement execution count for numRa values of 2, 3, and *n* in general.

So, the effect of a doubly nested loop is to give a cubic execution time as a function of the size of the matrices to be multiplied. The implication for execution time estimates is that if you double the size of the arrays you multiply the execution time by eight.

Statement	numRa **= 2**	numRa **= 3**	numRa **= *n***
(1)	3	4	$n + 1$
(2)	6	12	$(n + 1) \cdot n$
(3)	4	9	n^2
(4)	12	36	$n^2 \cdot (n + 1)$
(5)	8	27	n^3
(6)	4	9	n^2
Total:	37	97	$2n^3 + 4n^2 + 2n + 1$

Figure 18.9
Statement execution count for the procedure Multiply in Figure 18.8.

Using two-dimensional arrays

In Component Pascal, the declaration

```
ARRAY L1, L2 OF T
```

is really an abbreviation of

```
ARRAY L1 OF
   ARRAY L2 OF T
```

That is, a two-dimensional array is a one-dimensional array of vectors.

Example 18.1 In Figure 18.3, the declaration

```
Matrix = ARRAY 32, 32 OF REAL;
```

could be written

```
Matrix = ARRAY 32 OF ARRAY 32 OF REAL;
```

Considered in this light, variable matrix has 32 elements from matrix[0] up to matrix[31], each of which is a vector of 32 reals as Figure 18.10 shows. ▮

Consistent with this abbreviation, the notation matrix[i, j], which denotes the element in row i, column j of the matrix, is simply an abbreviation for matrix[i][j], which denotes element j in the vector matrix[i] as Figure 18.10 shows.

Example 18.2 The assignment statement max := mat[row, j] in Figure 18.6 can be written max := mat[row][j]. ▮

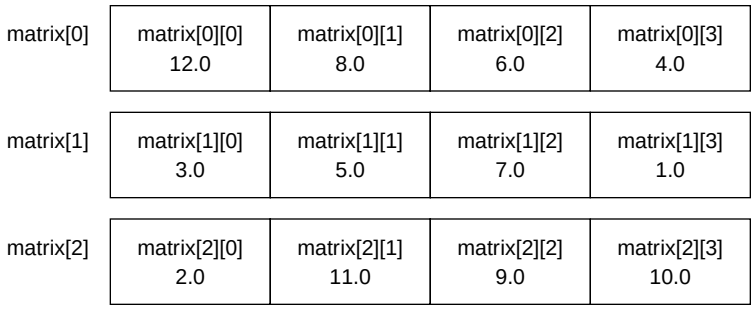

Figure 18.10
The matrix of Figure 18.3 considered as an array of vectors.

Exercises

1. Assume that matrices a and b of Figure 18.8 each have *n* rows and *n* columns. If it takes 50 μs for procedure Multiply to execute with $n = 25$, how many microseconds will it take to execute with $n = 65$? Use the approximate ratio where you neglect the low-order terms of the polynomial.

2. Assume that matrix a of Figure 18.8 has *l* rows and *m* columns, and matrix b has *m* rows and *n* columns. Write an expression in terms of *l*, *m*, and *n* for the statement execution count of procedure Multiply.

Problems

3. Write a program that inputs a two-dimensional array of real values from a window, and outputs to a new window the row sum of each row, the column sum of each column, and the grand total. For example if the focus window contains

```
 4.0  –6.0   1.0   3.0
–2.0   3.0   7.0   2.0
 1.0   0.0   4.0   5.0
```

the new window should contain

```
 4.0  –6.0   1.0   3.0   2.0
–2.0   3.0   7.0   2.0  10.0
 1.0   0.0   4.0   5.0  10.0
 3.0  -3.0  12.0  10.0  22.0
```

Hint: If the original matrix contains 3 rows and 4 columns, store the column sums in the 4th row and row sums in the 5th column. Test your program with a dialog box similar to that in Figure 18.1. Activate your dialog box with a menu selection.

4. Write a program to output to the Log the indices of the largest element of a two-dimensional array of real values. For example, if the input is identical to that of Problem 3, the output to the Log should be

```
Largest value: 7.00
Row: 1
Column: 2
```

Test your program with a dialog box similar to that in Figure 18.1. Activate your dialog box with a menu selection.

5. Declare

```
PROCEDURE Normalize (VAR mat: ARRAY OF ARRAY OF REAL; normR, numC: INTEGER)
```

where normR is a row in mat to be normalized, and numC is the number of columns in mat. Implement appropriate preconditions with the ASSERT statement. To normalize a row, divide every element in that row by the element in the row with the largest absolute value. For example, to normalize row 0 of the matrix in Problem 3, you would call

```
Normalize (matrix, 0, 4)
```

which would divide each element in the first row by –6.0, producing

–0.667	1.000	–0.167	–0.500
–2.000	3.000	7.000	2.000
1.000	0.000	4.000	5.000

To normalize row 1, the procedure should divide each element in the second row by 7.0, producing

4.000	–6.000	1.000	3.000
–0.286	0.429	1.000	0.286
1.000	0.000	4.000	5.000

Display the normalized matrix with three places past the decimal point. Test your procedure with a dialog box similar to that of Figure 18.4 with an input field for the user to enter the number of the row to normalize. When the user clicks the compute button, verify that the row number entered in the dialog box is valid. If it is, output the matrix before and after the normalization. Otherwise output an error message. Do not include any output statements in Normalize. Activate your dialog box with a menu selection.

6. Declare

```
PROCEDURE SwapRow (VAR mat: ARRAY OF ARRAY OF REAL; row1, row2, numC: INTEGER)
```

where numC is the number of columns in mat. Implement appropriate preconditions with the ASSERT statement. The procedure should exchange row1 with row2. Test your procedure with a dialog box similar to that of Figure 18.4 with input fields for the user to enter two row numbers. When the user clicks the compute button, verify that the row numbers entered in the dialog box are valid. If they are, output the matrix before and after the exchange. Otherwise output an error message. Do not include any output statements in SwapRow. Activate your dialog box with a menu selection.

7. Declare

```
PROCEDURE Transpose (VAR mat: ARRAY OF ARRAY OF REAL; VAR numR, numC: INTEGER)
```

where numR and numC are the number of rows and columns in mat. Implement appropriate preconditions with the ASSERT statement. The procedure should transpose mat and switch the values of numR and numC. To transpose a matrix, turn its columns into rows and its rows into columns. For example, the transpose of the matrix in Problem 3 is the following matrix with four rows and three columns.

```
 4.0  –2.0  1.0
–6.0   3.0  0.0
 1.0   7.0  4.0
 3.0   2.0  5.0
```

Test your procedure with a dialog box similar to that of Figure 18.1. When the user clicks the compute button, output the matrix before and after the transposition. Do not include any output statements in Transpose. Activate your dialog box with a menu selection.

8. Declare

```
PROCEDURE InitUnit (OUT mat: ARRAY OF ARRAY OF REAL; numR, numC: INTEGER)
```

where numR and numC are the number of rows and columns in mat. Implement appropriate preconditions with the ASSERT statement. The procedure should initialize the matrix to all zeros, except for ones on the diagonal. For example, if numR is 3 and numC is 4, mat should get

```
1.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0
0.0  0.0  1.0  0.0
```

Test your procedure with a dialog box that requests the user to enter the number of rows and columns. When the user clicks the compute button, verify that the number of rows and columns are each nonnegative. If they are, initialize the matrix and output it to a new window with one place past the decimal point. Otherwise, output an error message. Your procedure should work with the empty matrix, in which the number of rows or columns is zero. Do not use any output statements in InitUnit. Activate your dialog box with a menu selection.

9. Declare

```
PROCEDURE InitBand (OUT mat: ARRAY OF ARRAY OF REAL; numR, numC: INTEGER)
```

where the meanings of the parameters are the same as in Problem 8. Implement appropriate preconditions with the ASSERT statement. The procedure should initialize the matrix to all zeros, except for ones on the diagonal and elements immediately adjacent to the diagonal. For example, if numR is 4 and numC is 5, mat should get

```
1.0  1.0  0.0  0.0  0.0
1.0  1.0  1.0  0.0  0.0
0.0  1.0  1.0  1.0  0.0
0.0  0.0  1.0  1.0  1.0
```

Test your procedure as specified in Problem 8.

10. Generalize Chapter 15, Problem 27 to a two-dimensional array. Declare

```
PROCEDURE InitRandom (OUT mat: ARRAY OF ARRAY OF INTEGER; numR, numC: INTEGER)
```

that initializes the array to a random sequence of nonrepeating integers. Implement appropriate preconditions with the ASSERT statement. Test your procedure with a dialog box that inputs the number of rows and columns in a dialog box and gives the user the option to set the seed. When the user clicks the compute button, verify that the number of rows and columns are each nonnegative. If they are, initialize the matrix and output it to a new window. Otherwise, output an error message. For example, if the user enters 3 rows and 4 columns the procedure should initialize mat to

```
 0   1   2   3
 4   5   6   7
 8   9  10  11
```

then exchange mat[0, 0] with another component chosen at random, mat[0, 1] with another component chosen at random, and so on. Output the matrix of nonrepeating random integers to a new window. Do not use any output statements in InitRandom. Activate your dialog box with a menu selection.

11. Each integer of a two-dimensional array of integers represents the elevation at one point of some rugged terrain. A local maximum is a point whose integer value is greater than the values of its surrounding eight neighbors. For example, the integer array

```
20  30  30  43  53  72  83
41  40  53  61  77  95  99
42  62  90  85  71  87  88
30  60  70  50  49  56  58
```

has a local maximum at row two, column two because 90 is greater than 40, 53, 61, 62, 85, 60, 70, and 50. Values on the borders, such as 99, are not candidates for local maxima. Note also that 95 is not a local maximum because 99 is greater than 95. Write a program that inputs a two-dimensional array of integers with a dialog box similar to that of Figure 18.1. When the user clicks the compute button, output the location of all the local maxima, if any, to the Log.

12. A saddle point is a point whose integer value is greater than its two neighbors' values in the same row but smaller than its two neighbors' values in the same column. A point can also be a saddle point if its integer value is smaller than its two neighbors' values in the same row but greater than its two neighbors' values in the same column. For example, the integer array

```
48  52  30  43  67
64  55  50  58  95
61  62  40  51  70
56  60  32  48  49
```

has a saddle point at row one, column two because 50 is greater than 30 and 40, but less than 55 and 58. Values on the borders are not candidates for saddle points. Write a program that inputs a two-dimensional array of integers with a dialog box similar to that of Figure 18.1. When the user clicks the compute button, output the location of all the saddle points, if any, to the Log.