Complex Arrays Made Simple

Mary McDonald, PaineWebber Incorporated

Abstract

Many novice SAS programmers avoid arrays because they think that they are difficult. The truth is that they make work easier. This tutorial is a simple introduction to working with multidimensional and nested arrays. Topics covered are: common applications, loading constants, referencing, defining and processing.

Introduction

Multidimensional arrays are useful when you want to group your data into a table like arrangement. You know you need one when you start thinking that you could process your data in EXCEL a lot easier than in SAS. They reduce the amount of code that you have to write, make your program look tidier and make it easier to think about what you are doing.

Referencing and manipulation is not much different than for one-dimensional arrays. Actually the simple array is a special case of the multi-dimensional array. Neither kind will improve processing efficiency, but they will reduce typing. They also come in both implicit and explicit flavors.

The simple numeric array is defined as follows:

```
ARRAY array-name (size) array-elements;
```

Another to think about it is:

```
ARRAY ROW1 (size) columns-first-to-end;
                  OR
ARRAY SIMPLE (4) R1C1 R1C2 R1C3 R1C4;
```

The data would look like this:

	COL1	COL2	COL3	COL4
ROW1	240	1000	72	95

Adding another dimension just adds additional rows to your table:

	COL1	COL2	COL3	COL4
ROW1	240	1000	72	95
ROW2	341	998	84	83
ROW3	549	800	102	77

```
ARRAY DOUBLE(3,4) R1C1 R1C2 R1C3 R1C4
                  R2C1 R2C2 R2C3 R2C4
                  R3C1 R3C2 R3C3 R3C4
                  R3C1 R3C2 R3C3 R3C4;
```

Think of the first dimension as rows, the second is columns. For higher dimensions, just add another comma and the size of the next dimension. This goes in front of the row subscript, which is sort of odd at first. But it will be the outer most of the nested loops so it makes sense because SAS fills the array starting with the first dimension. 3-d and up arrays are rare, I have never needed one or seen one in 'real world' code.

LEVEL1:

	COL1	COL2	COL3	COL4
ROW1	240	1000	72	95
ROW2	341	998	84	83
ROW3	549	800	102	77

LEVEL2:

	COL1	COL2	COL3	COL4
ROW1	761	1022	78	16
ROW2	341	968	584	583
ROW3	549	780	109	76

```
ARRAY TRIPLE(2,3,4)
           L1R1C1 L1R1C2 L1R1C3 L1R1C4
           L1R2C1 L1R2C2 L1R2C3 L1R2C4
           L1R3C1 L1R3C2 L1R3C3 L1R3C4
           L1R3C1 L1R3C2 L1R3C3 L1R3C4
           L2R1C1 L2R1C2 L2R1C3 L2R1C4
           L2R2C1 L2R2C2 L2R2C3 L2R2C4
           L2R3C1 L2R3C2 L2R3C3 L2R3C4
           L2R3C1 L2R3C2 L2R3C3 L2R3C4
```

Processing is usually done in nested loops, the first dimension is the outer most and 'columns' are the inner most.

```
DO LEVEL = 1 TO 2;
     DO ROW =1 TO 3;
          DO COLUMN 1 TO 4;
           [Code to process the
            triple array data]
          END;
     END;
END;
```

DO UNTIL (evaluate the condition at the bottom of the loop), DO WHILE (evaluate at the top) and BY(to increment the index by a specified amount) work exactly the same way on all dimensions of arrays. DIM returns the first dimension by default. For 2-d plus specify the number desired. In our example, DIM(TRIPLE) would get the number of levels, DIM(TRIPLE,2) or DIM2(TRIPLE) would get the number of rows, DIM(TRIPLE,3) or DIM3(TRIPLE) the number of columns. Boundary specifications can be added to the subscript. LBOUND AND HBOUND will work just like DIM.

<$> and <length> for arrays of character variables works exactly the same. <*> doesn't work because SAS can not determine the array subscripts by counting the number of elements in multidimensional or *_TEMPORARY_* arrays.

Note:
If you are going to do exactly the same thing to all the array elements you can define the array as a one dimensional array and process inside a single loop.

```
ARRAY SIMPLE (24)
           L1R1C1 L1R1C2 L1R1C3 L1R1C4
           L1R2C1 L1R2C2 L1R2C3 L1R2C4
           L1R3C1 L1R3C2 L1R3C3 L1R3C4
           L1R3C1 L1R3C2 L1R3C3 L1R3C4
           L2R1C1 L2R1C2 L2R1C3 L2R1C4
           L2R2C1 L2R2C2 L2R2C3 L2R2C4
           L2R3C1 L2R3C2 L2R3C3 L2R3C4
           L2R3C1 L2R3C2 L2R3C3 L2R3C4;
```

Processing would be done in one loop:

```
DO J= 1 TO 24;
    [Code to process data]
END;
```

More complex processing requires complex arrays. This is easier to understand when you work through coding problems that are best solved with multidimensional or complex arrays. The following examples are based on real applications.

Problem 1:
This is part of a set of programs that process data on leased office space for the City of New York. The problem was to project the current costs for rent, utilities and renovation 5 years into the future. It was assumed that cost increase percentages would stay the same for those years: 3 percent increases for utilities, 5 percent for renovation and miscellaneous and 8 for rent.

Example 1:
This can be coded with multiple simple arrays as follows:

```
                    /* Property Costs */
ARRAY BASE  (5) RENT  ELEC  RENOV
                HVAC  MISC ;
ARRAY ONE   (5) RENT1 ELEC1 RENOV1
                HVAC1 MISC1 ;
ARRAY TW0   (5) RENT2 ELEC2 RENOV2
                HVAC2 MISC2 ;
ARRAY THREE (5) RENT3 ELEC3 RENOV3
                HVAC3 MISC3 ;
ARRAY FOUR  (5) RENT4 ELEC4 RENOV4
                HVAC4 MISC4 ;
ARRAY FIVE  (5) RENT5 ELEC5 RENOV5
                HVAC2 MISC2 ;
                     /* Annual Totals */
ARRAY TOTAL (5) TOTAL1-TOTAL5;
                      /* Constants */
ARRAY INCREASE (5) INC1-INC5;

RETAIN INC1 1.08 INC2 INC3 1.03 INC4
       INC5 1.05;
                       /* FIRST YEAR */
DO J = 1 TO 5;
     ONE(J) = BASE(J)*INCREASE(J);
     TOTAL1 + ONE(J);
END;
                      /* SECOND YEAR */
DO J = 1 TO 5;
     TWO(J) = ONE(J)*INCREASE(J);
     TOTAL2 + TWO(J);
END;
                       /* THIRD YEAR */
DO J = 1 TO 5;
     THREE(J) = TWO(J)*INCREASE(J);
     TOTAL3 + THREE(J);
```

```
END;
                        /* FOURTH YEAR */
DO J = 1 TO 5;
     FOUR(J) = THREE(J)*INCREASE(J);
     TOTAL4 + FOUR(J);
END;
                         /* FIFTH YEAR */
DO J = 1 TO 5;
     FIVE(J) = FOUR(J)*INCREASE(J);
     TOTAL5 + FIVE(J);
END;
DROP INC1-INC5;
```

That code will work and it is certainly a lot better than writing out each computation. However it is repetitive and we can certainly do better.

Example 2

Adding another loop and an 'array of arrays' will improve the code somewhat. Using a *_TEMPORARY_* ARRAY is a better way to code the array of increases. These are constants that aren't needed after this step. This is more efficient and the drop statement is no longer needed.

```
                    /* IMPLICIT ARRAYS */
ARRAY BASE  (J) RENT  ELEC  RENOV
                HVAC  MISC ;
ARRAY ONE   (J) RENT1 ELEC1 RENOV1
                HVAC1 MISC1 ;
ARRAY TW0   (J) RENT2 ELEC2 RENOV2
                HVAC2 MISC2 ;
ARRAY THREE (J) RENT3 ELEC3 RENOV3
                HVAC3 MISC3 ;
ARRAY FOUR  (J) RENT4 ELEC4 RENOV4
                HVAC4 MISC4 ;
ARRAY FIVE  (J) RENT5 ELEC5 RENOV5
                HVAC2 MISC2 ;
ARRAY TOTAL (K) TOTAL1-TOTAL5;

                          /* CONSTANTS */
ARRAY INCREASE (5) _TEMPORARY_
( 1.08 1.03 1.03 1.05 1.05);

                    /* ARRAY OF ARRAYS */
ARRAY YEARS (K) ONE TWO THREE
                FOUR FIVE;
ARRAY PREV  (K) BASE ONE TWO THREE
                FOUR ;

                        /* CALCULATIONS */
DO K = 1 TO 5;
   DO J = 1 TO 5;
      YEARS = PREV*INCREASE;
      TOTAL + YEARS;
   END;
END;
```

This is code that SAS Institute doesn't want you to use, these are 'implicit arrays'. Prior to SAS 6.06 this was the only way to do multi dimensional arrays. The index variable is specified when you create the array and is not needed when you reference it. Since they both have the same index variable you can process them in the same loop and you can also process the TOTALS array.

Example3

SAS Institute recommends that you use multidimensional arrays instead of the nested implicit arrays in the previous example because they are more efficient. There is certainly less code.

```
ARRAY COSTS (6,5)
    RENT  ELEC  RENOV  HVAC  MISC
    RENT1 ELEC1 RENOV1 HVAC1 MISC1
    RENT2 ELEC2 RENOV2 HVAC2 MISC2
    RENT3 ELEC3 RENOV3 HVAC3 MISC3
    RENT4 ELEC4 RENOV4 HVAC4 MISC4
    RENT5 ELEC5 RENOV5 HVAC5 MISC5;

                            /* CONSTANTS */
ARRAY INCREASE (5) _TEMPORARY_
( 1.08 1.03 1.03 1.05 1.05);

                        /* CALCULATION */
DO R = 2 TO 6;
    P = R – 1;
    DO C = 1 TO 5;
     COSTS(R,C)=COSTS(P)*INCS(C);
       TOTAL(R) + COSTS(R,C);
    END;
END;
```

I started the loop for rows with '2' for the second which is the first one we project, the first is the base costs. The 'P' index is used to reference the prior year's row.

Problem 2

The application needs to calculate bonus amounts that are paid based on a combination of points earned for opening new accounts and increases in assets in all client accounts. Each broker has an individual set of eight asset levels.

The seven points categories for new accounts are the same for every one. Those who do not meet both asset and new account goals get nothing. There is no way to compute the amounts to be paid other than looking them up on the table. It could be coded using seven separate arrays one for each level of points but there is a neater solution.

The first step is to code a value for the points ranges using PROC FORMAT.

```
PROC FORMAT;
value ptscd
     LOW-<20.00    = '0'
     20.00-<30.00  = '1'
     30.00-<40.00  = '2'
     40.00-<50.00  = '3'
     50.00-<60.00  = '4'
     60.00-<70.00  = '5'
     70.00-<80.00  = '6'
     80.00-high    = '7' ;
```

The individual asset goals are read into an array from an ordinary flat file.

```
DATA GOALS;
     INFILE GOALS MISSOVER;
     ARRAY GOAL (8);
     INPUT @1 SSN @;
     N = 10;
     DO J=1 TO 8;
       @N GOAL(J) 12. @;
        N +12;
     END;
```

Because I did not list the variables in the GOAL ARRAY, SAS will name them GOAL1 to GOAL8 for me. It is usually better to list them so you control what their names are. In this case I only need them to do the calculation. If it were all one data step I would use a temporary array.

```
DATA FATOTALS;
     MERGE FATOTALS(IN=F) GOALS(IN=G);
     BY SSN;
     IF F AND G;

                     /* BONUS AMOUNTS */
     ARRAY BONUS (0:7,0:8) _temporary_
(0    0    0    0    0    0    0    0    0
 0 1000 1500 2000 2500 3000 3500  4000  4500
 0 1750 2250 2750 3250 3875 4500  5125  5750
 0 2500 3000 3500 4000 4750 5500  6250  7000
 0 3250 3750 4250 4750 5500 6250  7000  7750
 0 4000 4500 5000 5500 6250 7000  7750  8500
 0 4750 5250 5750 6250 7125 8000  8875  9750
 0 5500 6000 6500 7000 8000 9000 10000 11000)
;
```

I define the bounds of the array subscripts to start at '0' rather than '1', which is the default. In this case it happens to be easier for me to code the asset levels. SAS is supposed to work more efficiently if do loops and subscripts start at '0', but most programmer's minds don't. I am also going to process the GOAL array 'backwards' stopping at the highest level met.

```
                       /* Asset Goals */
ARRAY GOAL (8)   GOAL1-GOAL8;
                     /* Initial Level */
LEVEL = 0;
                     /* Compute Level */
DO J=8 TO 1 BY -1 UNTIL(LEVEL=J);
    IF ASSETS GE GOAL(J) THEN LEVEL=J;
END;
                /* FIND POINTS RANGE */
PTS_CAT = PUT(POINTS,PTSCD.)*1;
                      /* FIGURE BONUS */
BONUS = BONUS(PTS_CAT,LEVEL);

DROP GOAL1-GOAL8 J ;
RUN;
```

Conclusion:

Multidimensional arrays may not be used frequently but when you need one it really helps to have that tool available. Bob Virgil in his paper *Introduction to Arrays*, made the point that "In most applications which use arrays, arrays are 10% of the pie and other tools make up 90%." Of that 10%, 90% are one-dimensional simple arrays, but that other 10% has all the fun!

Contact:

Mary McDonald
Database Marketing and Analysis
PaineWebber Incorporated
1200 Harbor Blvd, 6th floor
Weehawken, NJ
Mmcdonal@painewebber.com