

USING SAS* ARRAYS

Eric Webster, Bradford Exchange USA Ltd.

WHAT ARE ARRAYS?

Arrays are a way of referring to a group of variables in one observation by a single name.

WHY WOULD I USE ARRAYS??

Arrays are useful for a variety of purposes:

- * Performing repetitive calculations on a large number of variables, such as scaling by 10;
- * Restructuring data sets;
- * Initializing variables to a particular value, or particular set of values;
- * Collecting simple frequency counts while passing very large data sets;

EXAMPLE 1: SQUARING VARIABLES

In this example, each variable in the data set needs to be squared (multiplied by itself). There are 25 variables in each observation. The simple solution:

```
DATA EXAMPLE1;
  SET MYLIB.EXAMPLE1;
  ARRAY MYARRAY(25) VAR1 - VAR25;
  DO I = 1 TO 25;
    MYARRAY(I) = MYARRAY(I) * MYARRAY(I);
  END;
  RUN;
```

The variables VAR1 through VAR25 are collected into the group "MYARRAY", which is a 25 element array. The DO-group passes over each of the 25 variables, multiplies it by itself, and stores the result in the original variable. The variable "I" is used as a subscript, or placeholder, to tell SAS which variable to work on.

An array is just shorthand for the variables in the variable list. for example, if VAR1 through VAR4 look like:

VARIABLE:	VAR1	VAR2	VAR3	VAR4
VALUE:	7	0.25	-101	714

Then the array represents the following:

ELEMENT:	MYARRAY(1)	MYARRAY(2)	MYARRAY(3)	MYARRAY(4)
VARIABLE:	VAR1	VAR2	VAR3	VAR4
VALUE:	7	0.25	-101	714

DECLARING ARRAYS

Arrays must be declared before they can be used, and the declaration only lasts for the duration of the DATA step in which they are defined. If the same array is to be used in many data steps, the declarations must be copied into each data step. Alternatively, a simple macro can be written to generate the declarations, and invoked in each data step.

The basic declaration has several parts. take the following example:

```
ARRAY MYARRAY      (17)  $      NAME1-NAME17;  
  A      B      C      D      E
```

- | | |
|---------------------|---|
| A. ARRAY STATEMENT | - Required |
| B. ARRAY NAME | - Must be a valid SAS name, and cannot conflict with any other name |
| C. ARRAY BOUNDS | - Specifies how many variables are to be in the array. Can be given as (*), in which case SAS will determine the number of variables based on the variable list |
| D. CHARACTER SYMBOL | - If the array consists of character variables, the "\$" is placed here |
| E. VARIABLE LIST | - Can be any collection of variables of the same type (numeric or character). If the variables do not already exist, they are created automatically (as character if the "\$" is present) |

The following are valid array declarations:

```
ARRAY BIGARRAY(1000)  VAR1-VAR1000;  
ARRAY HOWMANY(*)      VAR1-VAR20 NUM7-NUM9 VAL COUNT LIST8;  
ARRAY SPOUSE(*)       $ BOB CAROL TED ALICE;
```

EXAMPLE 2: RESTRUCTURING A DATASET

One of the most useful applications of arrays is dataset restructuring. Consider the following example: you work for a retail chain, and have a dataset consisting of total store sales by month for 1989 for each store. The variables are SALES1-SALES12, where SALES1 = January sales for that store, SALES2 = February sales for that store, etc. There is one observation for each store in the chain. You need to produce a line graph with three lines, one for the minimum sales by store for each month, one for the mean sales per month, and one for the maximum sales per month.

You decide to use PROC MEANS to give you the statistics in a new dataset:

```
PROC MEANS NOPRINT;  
  VAR SALES1 - SALES12;  
  OUTPUT  
    OUT = MEANSET  
    MEAN = MSALE1-MSALE12  
    MAX  = HI1-HI12  
    MIN  = LO1-LO12;  
RUN;
```

This new dataset, MEANSET, has one observation, 36 variables, and needs to be rebuilt for PROC PLOT:

```
DATA PLOTSET;
  SET MEANSET;
  ARRAY MEANS(12) MSALE1-MSALE12;
  ARRAY LO(12)      LO1-LO12;
  ARRAY HI(12)      HI1-HI12;
  DO MONTH = 1 TO 12;
    LOW = LO(MONTH);    HIGH = HI(MONTH);
    MEAN = MEANS(MONTH); OUTPUT;
  END;
  KEEP LOW HIGH MEAN MONTH;
  RUN;
```

PLOTSET has four variables with 12 observations, ready to be plotted:

```
PROC PLOT;
  PLOT (LOW MEAN HIGH) * MONTH / OVERLAY;
  RUN;
```

MULTIDIMENSIONAL ARRAYS

All of the arrays examined so far are "single dimensional", or in other words have one subscript (such as MEANS(5)). It is possible to use arrays with multiple subscripts, e.g. SALES(1,12); these are called multidimensional arrays.

Multidimensional arrays are declared by giving the number of variables for each dimension. The total number of variables is then the product of the upper bounds. For example:

```
ARRAY TABLE(3, 4) VALUE1-VALUE12;
```

Twelve variables are specified because there are $3 \times 4 = 12$ elements in the array. TABLE then looks like:

TABLE(1,1)	TABLE(1,2)	TABLE(1,3)	TABLE(1,4)
TABLE(2,1)	TABLE(2,2)	TABLE(2,3)	TABLE(2,4)
TABLE(3,1)	TABLE(3,2)	TABLE(3,3)	TABLE(3,4)

The rightmost subscript stands for columns, and the next subscript to the left is for rows. Arrays can also be defined to have many dimensions, e.g., NUMBER(1,4,7,2) has four dimensions.

If VALUE1 = 1, VALUE2 = 2, VALUE3 = 3, ..., VALUE12 = 12, then

```
TABLE(1,1) = VALUE1 = 1
TABLE(1,2) = VALUE2 = 2
TABLE(1,3) = VALUE3 = 3
TABLE(1,4) = VALUE4 = 4
TABLE(2,1) = VALUE5 = 5
TABLE(2,2) = VALUE6 = 6

TABLE(3,4) = VALUE12 = 12
```

READING DATA INTO MULTIDIMENSIONAL ARRAYS

As with any array, data can be read in to either the variables in the variable list or using the array name itself in the input statement. Both accomplish the same purpose - reading data into a variable in the variable list automatically makes that element of the array available, and similarly reading data into an element of an array gives the corresponding variable its value. Below are three examples of how this is done.

A. ALL VALUES IN ONE OBSERVATION: VARIABLE LIST

If all twelve data elements are in one observation, the easiest way to read in the array is to use the variable list:

```
INPUT VALUE1-VALUE12;
```

B. ALL VALUES IN ONE OBSERVATION: ARRAY NAME

If all twelve data elements are in one observation, and you wish to read the data using the array name, then the input statement would look like:

```
DO ROW = 1 TO 3;
  DO COLUMN = 1 TO 4;
    INPUT TABLE(ROW,COLUMN) @;
  END;
END;
INPUT;
```

The trailing "@" prevents SAS from reading the next observation, and the final input releases the input record.

C. VALUES IN DIFFERENT OBSERVATIONS: ARRAY NAME

More commonly your array definition will look like your input file, which in this case would be three observations of four variables. The code here is more complicated:

```
DATA INFO;
  ARRAY TABLE(3,4) VALUE1-VALUE12;
  RETAIN VALUE1-VALUE12;
  DO COLUMN = 1 TO 4;
    INPUT TABLE(_N_, COLUMN) @;
  END;
  INPUT;
  IF _N_ = 3 THEN OUTPUT;
  RUN;
```

Here we let the automatic variable `_N_` keep track of the row number. The `RETAIN` is necessary, because otherwise when the next observation is read all 12 variables would be set to missing. The final test against `_N_` forces SAS to only produce one observation; in a more realistic situation you might use an `END=` option on your input file instead of a direct test of `_N_`.

EXAMPLE 3: USING A 2-DIMENSIONAL LOOKUP TABLE

Assume you work for an auto insurance company, and have just completed a risk analysis based on number of children in the family and number of accidents in the past 3 years. The number of children is broken down into 1, 2, 3, 4, or 5+; the number of accidents in the past 3 years is broken down into 1, 2, 3, or 4+. For each combination you have computed a premium adjustment factor; for example, someone with two children and has had 3 accidents in the last three years might have a factor of 1.4, which would turn a premium of \$400 to $1.4 \times \$400 = \560 .

Your factor table is stored in file `FACTORS`, and you would like to read it into an array so that you can process the next set of billing statements. Each person has on file the number of children and number of accidents, stored in dataset `DATABASE.HISTORY`.

The lookup table is stored in array TABLE. The following code reads in the table, then processes the data into a new dataset:

```
DATA BILLING;
  ARRAY TABLE(5, 4) FACTOR1-FACTOR20;
  RETAIN FACTOR1-FACTOR20;
  IF _N_ = 1 THEN DO;
    FILE FACTORS;
    INPUT FACTOR1-FACTOR20;
  END;
  SET DATABASE.HISTORY;
  IF CHILDREN > 5 THEN CHILDREN = 5;
  IF ACCIDENT > 4 THEN ACCIDENT = 4;
  ADJPREM = PREMIUM * TABLE(CHILDREN, ACCIDENT);
  DROP FACTOR1-FACTOR20;
RUN;
```

The DROP statement at the end prevents the factors from being added to each new observation.

EXAMPLE 4: COUNTS FROM VERY LARGE DATASETS

A very useful application of arrays is to produce simple counts from very large datasets. For example, consider passing a 10-million entry sequential file to determine how many people on file 65 years or older with more than one home live in each state. Since the data is not in a SAS dataset, a simple FREQ with a WHERE statement (version 6) is not straightforward, and is not possible at all with version 5.

One approach to this problem would be to run a data step with a subsetting IF to get the right people, and perform a PROC FREQ; however, this creates a copy of the data, which can use far more computer resources than are available.

An array can come to the rescue here. Run a data step on the database, use a subsetting IF, and keep counters by state in your array. The STFIPS function translates the state code stored on the database into a number usable as an array index (the FIPS code, which ranges from 1 to 79); finally, do not output the counts until all the data has been passed. The net effect is that no additional storage is utilized to produce the counts, and the data must only be passed once.

```
DATA COUNTSET;
  ARRAY COUNTS(79) COUNT1-COUNT79;
  RETAIN COUNT1-COUNT79;
  LENGTH STATE $20.0;
  INFILE DATABASE END=DONE;
  INPUT <----->;
  IF (AGE GE 65) AND (HOMES > 1);
  FIP = STFIPS(STATE);
  COUNTS(FIP) = COUNTS(FIP) + 1;
  IF DONE THEN
    DO I = 1 TO 79;
      STATE = FIPNAME(I);
      PEOPLE = COUNTS(I);
      OUTPUT;
      KEEP STATE PEOPLE;
    END;
RUN;
```

THE DIM AND DIMn FUNCTIONS

If the array you are using has been defined with the (*) array bounds (meaning you let SAS figure out how many elements are in the array) or the array definition changes frequently, DO-loops can be difficult to code. SAS provides functions DIM and DIMn to provide a convenient means of coding loops. Both functions accept arrays as

input, and return the number of elements in either the one-dimensional array (DIM) or in the nth dimension of a multidimensional array (DIMn). For example:

```

ARRAY LITTLE(*)  A B C D E F;
ARRAY BIG(3,2)   A1 - A6
gives
DIM(LITTLE)      = 6
DIM1(BIG)        = 3
DIM2(BIG)        = 2

```

and

```

DO I = 1 TO DIM(LITTLE);
  PUT LITTLE(I);
END;

```

SPECIAL BOUNDS

When defining an array each subscript is assumed to begin at one and go to the upper bound of the array or of that dimension. This default can be overridden by specifying both the upper and lower bounds for each dimension, separated by a colon:

```

ARRAY SALES(78:89)   YEAR78 - YEAR89;
ARRAY SCORE(1:3, 30:39) TEST1-TEST30;

```

In this case SALES can only be referred to with a subscript in the proper range: SALES(85).

INITIAL VALUES

Another useful tool arrays provide is the automatic setting of initial values. If a set of values follows the variable list in an array definition, all of the variables are set to that initial value. This feature can make it useful to define an array without ever actually using it, just to force a set of initial values. For example:

```
MYARR(5) VALUE1-VALUE5(-1 -1 -1 -1 -1);
```

All 5 elements of MYARR are set to -1 on each iteration of the data step. If more array elements are specified than initial values, then the remaining elements receive missing values.

IMPLICITLY SUBSCRIPTED ARRAYS

It is possible to define an array so that a subscript is not explicitly given, but rather a particular variable (index variable) is set to a value which represents the element of the array to use (implicit subscripting). Using explicit subscripts is preferred, but implicit subscripting is possible.

```
ARRAY MYARRAY(PTR) VAR1-VAR5;
```

Here PTR (the index variable) specifies the variable in a reference. The array is then referred to by itself:

```

PTR = 1; PUT MYARRAY;  /* RETURNS THE VALUE OF VAR1 */
PTR = 4; PUT MYARRAY;  /* RETURNS THE VALUE OF VAR4 */

DO PTR = 1 TO 5;
  PUT MYARRAY;         /* PUTS VAR1 - VAR5 */
END;

```

The opportunity for coding errors is much higher with implicitly subscripted arrays, particularly with several arrays using several index variables.

* SAS is the registered trademark of SAS Institute Inc., Cary NC, USA.