SAS Tutorial Session - Working with Arrays
Dr. Ronald Cody - Robert Wood Johnson Medical School

I. Introduction

SAS arrays are a facility that can reduce the amount of coding in a data step. Although often thought of as an advanced programming tool, there are many applications of arrays that can be easily mastered. This talk will demonstrate some of the more common uses of SAS arrays. In the simplest form, an array is a list of SAS variable names. We can use an array to replace multiple lines of SAS code where the only changes from line to line are the variable names used. The original implementation of ARRAYS prior to Version 5, used what is called implicit subscripting. We will discuss this older form of arrays later, but for now, we will concentrate on the explicit form.

II. Using an Array to substitute a SAS missing value for 999

We will start off with a simple example. We have a data set where a value of 999 was used to represent a missing value (probably coded by an SPSS programmer). Our goal is to change all values of 999 to the SAS missing value.

Here is a SAS data step that does not use an array:

```
DATA EX1;
INPUT ABC D E;
IF A=999 THEN A=.;
IF B=999 THEN B=.;
IF C=999 THEN C=.;
IF D=999 THEN D=.;
IF E=999 THEN E=.;
CARDS;
(data lines)
RUN;
```

Here is the same program, using arrays:

```
DATA EX1A;
ARRAY X[5] ABC D E;
DO I=1 TO 5;
   IF X[I]=999 THEN X[I]=.;
END;
DROP I;
CARDS;
(data lines)
RUN;
```

Without even knowing how arrays work, you can see that we did not do much to reduce the size of our program. However, if we had a hundred variables rather than five, the size differential would have been more obvious. Now to explain what we did. First, the form of the array statement we used is:

```
ARRAYS arrayname[n] sas_variables;
```

(Where n is the number of elements and sas_variables is a list of SAS variables)

The statement begins with the word ARRAY, followed by an array name. Valid array names meet the same criteria as SAS variable names. However, an array name may not be the same as the name of a variable in your data set. Following the arrayname, in square brackets (in PC/SAS, {} brackets may also be used), is the number of elements (variables) which the array is to represent. An asterisk "*" may be substituted for this number if you do not care to do the counting yourself. (Note: I almost always use the form [*] since I do not count too well.) Finally, following the number of elements, is a list of SAS variable names. The SAS variable name conventions _ and -- may be used as well as the reserved SAS names _NUMERIC_, _CHARACTER_, and _ALL_. It is important to note that except for the single notation, a list of variables must have been previously defined, either by placing the ARRAY statement after the input statement or some other mechanism (such as a LENGTH or RETAIN statement). In our example above, the array name X will represent the five variables A, B, C, D, and E. With this form of the array statement we can tell the data step which element of the array we want to process by using a subscript (also called an index). Thus, X[1] (the first element of the X array) is equivalent to the variable name A; X[2] will represent B and so forth. To make this equivalence clear, the two statements which follow are identical once our array has been defined.

```
SUM = A + B + D;
```

is equivalent to

```
```

We usually find it convenient to use our arrays in some sort of iterative process. Our first example uses a DO statement. The syntax of a DO loop is:
DO counter = start TO end BY increment;
(lines of SAS code)
END;

The SAS statements between the DO and END statement will be repeated according the directions specified in the DO statement. Our simple example used "I" as the counter, 1 as the start value, 5 as the end value, and no value specified for the increment (which then defaults to 1). So, any SAS lines of code between our DO and END statement will be repeated five times, with the value of "I" taking on values of 1, 2, 3, 4, and 5. We had the SAS statement

IF X[I]=999 THEN X[I]=.;

inside our loop. The first time through the loop, "I" has a value of 1. So the line of SAS code is:


Remember that this is equivalent to:

IF A=999 then A=.;

The next time through the loop, "I" will have a value of 2 and the line will be interpreted as:


which is equivalent to:

IF B=999 THEN B=.;

OK, do you have the picture? You can now see why the two programs above are equivalent. As a matter of fact, the best way to begin using arrays is to write out a few lines of SAS code without using arrays. Then, noticing that the lines all look the same except for the variable names, you write an ARRAY statement to represent your variables, substitute the array name for the variable name that changes in each line, and place the line in a DO loop so that the process will be repeated for each element in the array list. You should also notice in our first example, that we DROPPED the counter variable "I." If we forgot to do this, each observation in our data set would have an "I" variable with a value of six. Why six? Because DO loops increment the counter before they check if the resulting value is greater than the ENDING value, at which time the iteration stops. We will show you later how to generalize this program to change the missing value from 999 to . for all our numeric variables. But first, here is another good example of array processing.

III. English to Metric Conversion Example

In this example, we will convert some weights and heights from English units to metric units.

Program without arrays:

DATA EX2;
INPUT WEIGHT1-WEIGHT5 HEIGHT1-HEIGHT5;
WEIGHT1=WEIGHT1/2.2;
WEIGHT2=WEIGHT2/2.2;
WEIGHT3=WEIGHT3/2.2;
WEIGHT4=WEIGHT4/2.2;
WEIGHT5=WEIGHT5/2.2;
HEIGHT1=2.54*HEIGHT1;
HEIGHT1=2.54*HEIGHT2;
HEIGHT1=2.54*HEIGHT3;
HEIGHT1=2.54*HEIGHT4;
HEIGHT1=2.54*HEIGHT5;
CARDS;
(data goes here)
RUN;

Program with arrays:

DATA EX2A;
INPUT WEIGHT1-WEIGHT5 HEIGHT1-HEIGHT5;
ARRAY WEIGHT[*] WEIGHT1-WEIGHT5;
ARRAY HEIGHT[*] HEIGHT1-HEIGHT5;
DO J=1 TO 5;
WEIGHT[J]=WEIGHT[J]/2.2;
HEIGHT[J]=2.54*HEIGHT[J];
END;
DROP J;
CARDS;
(data goes here)
RUN;

Notice that we used an asterisk in place of the number of elements so we did not have to count. We defined an array for the weights and one for the heights. Please realize that the array names WEIGHT and HEIGHT are arbitrary. We could have named them FRED and GEORGE. However, it is convenient to use the above convention when naming arrays. It's easy to remember that the elements of the array WEIGHT are your WEIGHTn variables. Be sure, however, that you do not have a variable in your data set with the same name. In our example, we could not have also had variables called WEIGHT and HEIGHT.

IV. Using an ARRAY to Create New Variables

So far, our arrays have represented variables that were listed in the INPUT statement. This is not necessary. We can define an array with a list of variables that are not included as INPUT variables. We will modify the above program to demonstrate this. Instead of converting the original WEIGHTn and HEIGHTn variables to metric values, we want to create ten new variables which will have
metric values. Thus, we will be able to use either the English or metric variables in any procedures that follow. Here is the code:

DATA EXJA;
INPUT WEIGHT1-WEIGHT5 HEIGHT1-HEIGHT5;
ARRAY WEIGHT[*] WEIGHT1-WEIGHT5;
ARRAY HEIGHT[*] HEIGHT1-HEIGHT5;
ARRAY WTKG[*] WTKG1-WTKG5;
ARRAY HTCM[*] HTCM1-HTCM5;
DO J=1 TO 5;
  WTKG[J]=WEIGHT[J]/2.2;
  HTCM[J]=2.54*HEIGHT[J];
END;
DROP J;
CARDS;
(data goes here)
RUN;

The resulting data set will contain the original five weight and five height variables as well as the five metric weight and five metric height ones.

V. Replacing 999 with SAS Missing Values for All Numeric Variables

We now want to expand example 1. to replace missing values of 999 with SAS missing values for all numeric variables.

DATA EXSA;
INPUT A B C X1-X20;
ARRAY X[*] _NUMERIC_ ;
DO I=1 TO DIM(X);
  IF X[I]=999 THEN X[I]=.;
END;
DROP I;
CARDS;
(data lines)
RUN;

The first change is to substitute the SAS internal variable _NUMERIC_ for the variable list of the ARRAY statement. This internal SAS name is available anywhere in a SAS program, in either a DATA or PROC step. Note also that the internal name _CHARACTER_ is also available and, as you would expect, it refers to all character variables. We used an asterisk in the ARRAY statement since we did not want to count how many numeric variables there were in our data set. Since the number of array elements is unknown, we used the DIM function (DIM stands for Dimension) in the next line. This function, whose argument is an array name, in this case X, returns the number of elements in the array, in this case, the number of numeric SAS variables. The remainder of the program is identical to example 1. This is a good example of an array that can save a considerable amount of SAS coding, especially in a data set with a lot of numeric variables.

VI. Temporary Arrays

Even those of you who used arrays under version 5 of the SAS system may be unfamiliar with the array attribute _TEMPORARY_. A temporary array, as the name implies, has only temporary variables which can be referred to only as ARRAYNAME[index]. This can be of value when speed and memory are of concern. As an example, at the Robert Wood Johnson Medical School, we had a very large test scoring program which could unscramble multiple versions of a test before performing an item analysis. The mapping of each version to the main version was accomplished using temporary arrays. Using arrays with real variables would have added hundreds of variables to the data set and exceeded memory capacity on a PC. Here is an example where the elements of a temporary array are used to hold the passing scores on each of ten tests.

DATA TEST;
ARRAY X[10] _TEMPORARY_ (65 60 60 65 70 62 66 67 64 60);
*X[1] TO X[10] HOLD PASSING SCORES ON TESTS;
ARRAY SCORE[10] SCORE1-SCORE10;
INPUT 10 SCORE1-SCORE10;
NUMPASS=0;
DO I = 1 TO 10;
  IF SCORE[I] GE XII] THEN NUMPASS+1;
END;
DROP I;
CARDS;
001 90 60 60 90 90 90 90 90 90 61
002 40 50 60 70 80 90 80 70 60 50
RUN;
PROC PRINT NOOBS;
RUN;

In this example, the variables X[1] to X[10] exist only as subscripted elements of the array X. Notice that there are no Xn variables in the data set and none in the DROP statement. Whenever you need variables only in the data step (i.e. you would have DROPPED them anyway) and you want an array, use the _TEMPORARY_ attribute to save memory and time.

VII. Arrayname[*] Notation in a Data Step

A little used, obscurely documented, but very useful feature of arrays is the ability to refer to all elements of an array with the ARRAYNAME[*] notation. To see how this works, look at the following example:

DATA EX7A;
ARRAY X[*] A B C D E;
INPUT X[*];
S = SUM (OF X[*]);
CARDS;
(data lines)
RUN;

We used the asterisk notation both in the INPUT statement and as the argument of the SUM function.

VIII. Using the Array Index as a Data Set Variable

In this example, we will use an array where the index of the array becomes a useful variable in our data set. Let's look at the problem of transforming a data set where each SAS observation contains several measurements on each subject at different times. We want a program that will transform this data set into one where each observation contains data on one subject at a single time. In addition, this new data set should contain a variable (which we will call TIME) that indicates at which time the measure was taken. To clarify this, here is a diagram of the original data set:

<table>
<thead>
<tr>
<th>SUBJ</th>
<th>TIME</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where X1 is a measure taken at time=1, X2 is the same measure taken at time=2 etc. We want the new data set to look like this:

<table>
<thead>
<tr>
<th>SUBJ</th>
<th>TIME</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

We could program this without arrays like this:

DATA EX8;
SET oldname;
ARRAY XX[3] X1-X3;
DO TIME=1 TO 3;
  X=XX[TIME];
  OUTPUT;
END;
DROP X1-X3;
RUN;

Notice the efficiency of using TIME as the index of our DO loop, thus saving us the problem of setting a value for TIME manually. Also notice that the array could not have been named X since we used X as a variable in our data set. Obviously with more levels of TIME, this use of arrays becomes even more advantageous.

IX. Character Arrays

So far, we have made arrays of numeric variables. Character variable arrays are also possible. If a variable has already been identified as a character variable (in a LENGTH or INPUT statement for example), then an array can be set up just the same as the arrays of numeric variables we have already seen. However, if the array statement is placed in the DATA step before the variables have been defined (such as directly after the DATA statement), the array statement must include a $ after the array name to indicate that the variables being assigned to the array are to be character. As a good programming practice, you should include a $ for all character arrays. Here is an example of a character array:

DATA EX9A;
ARRAY X[*] $ A B C;
INPUT A $ 1-2 B $ 3-4 C $ 5-6;
DO I = 1 TO 3;
  IF X[I] = 'XX' THEN X[I] = ' ';
END;
DROP I;
CARDS;
(data lines)
RUN;

X. Initial Values and Attributes

Besides declaring an array as character, we can define the variable lengths and initial values with the ARRAY statement. The general form is:

ARRAY arrayname[n or *] $ length
list of vars (initial values);

For example, to declare an array of numeric variables to all be length 4, we would have:

ARRAY ABC[*] 4 X Y Z;
To assign initial values of 0, 1, and 3 to X, Y, and Z respectively, we would add:

```sas
ARRAY ABC[*] 4 X Y Z (0 1 3);
```

XI. Specifying Array Bounds

So far, all our array indices have run from 1 to n (where n is the number of variables in the variable list). There are times when it is useful to be able to specify lower and upper bounds on the array index. One of the best examples is one used in the SAS Language Guide. This example uses variable names such as cost88, cost89, and cost90 where the numeric part refers to the year. An array statement such as:

```sas
ARRAY COST[3] COST88-COST90;
```


```sas
ARRAY COST[88:90] COST88-COST90;
```

you have the advantage that COST[88] refers to 1988 and so forth.

XII. Multidimensional Arrays

SAS allows for multidimensional arrays. In the example that follows, the scores for five tests are kept for each of three classes. The first dimension will tell us which class the scores are for, the second, the test within class. So, on the data line, the first five scores are for class one, the second five for class two, and the third five for class three. A diagram of the data for one subject is shown in table I:

Here is the program using arrays:

```sas
DATA EX12A;
ARRAY X[3,5] X1-X15;
ARRAY AVE[3] AVE1-AVE3;
*NOTE FIRST DIMENSION IS CLASS NUMBER
SECOND IS TEST NUMBER (IE. 5 TESTS FOR EACH OF 3 CLASSES;
INPUT X[*] @;
DO CLASS=1 TO 3;
   DO TEST=1 TO 5;
      AVE[CLASS]+X[CLASS,TEST]/5;
   END;
END:
DROP TEST CLASS X1-X15;
CARDS;
90 90 100 100 95 80 80 80 90 90
70 70 70 60 70
(data for other students)
RUN;
```

The first DO loop used CLASS as its index. CLASS is first set to 1. Then while CLASS is held at 1, the inner loop (DO TEST) executes for all five tests. Inside this loop, the statement AVE[CLASS]+X[CLASS,TEST]/5 adds each of the five tests scores (divided by 5) so that after the inner loop has executed five times, AVE[CLASS] will be the average for one of the three classes. If you have not seen nested DO loops before, this may seem confusing. To help clarify this, Table II shows the actual 15 computations performed by the nested loops:

XIII. Implicitly Subscripted Arrays

Before we leave the topic of arrays, we should mention the alternate type of array which does not explicitly show the subscript when you refer to an array element in the data step. This was the original form of the array statement in version 5 that was superseded by the explicit subscript form that we have discussed until now. We strongly recommend that you use the explicit form. However, since the implicit form is still supported and you may have to maintain older SAS code that contains this type of array, we will briefly show you how it works. The form of the ARRAY statement is:

```sas
ARRAY arrayname(index variable) list of SAS variables;
```

Length and $ attributes are also available and placed before the list of SAS variables. When using the array name in a data step, the index variable is first set (usually by a DO loop) and the array name is used without a subscript. Here is an example:

```sas
DATA EX13A;
ARRAY MOZART(I) ABC D E;
INPUT ABC D E;$
DO I = 1 TO 5;
   IF MOZART=999 THEN MOZART=.;
END;
DROP I;
CARDS;
(data lines)
```

Notice that inside the DO loop, the array name MOZART is used without a subscript. When I=1, MOZART will represent the variable name A; when I=2, MOZART will represent B, and so forth. A default subscript _I_ will be used if no subscript is indicated in the ARRAY statement. The DO loop would then read "DO _I_ = 1 to 5." A very useful form of the DO loop, "DO OVER" is available when
the implicit subscript form of the array is used. DO OVER is equivalent to "DO subscript = lower bound TO upper bound" but saves us the trouble of using the DIM function or knowing how many elements are in the array. Furthermore, if you use the default subscript \_ it, you don't even have to remember to DROP it since it is an internal SAS variable and not a variable in the data set. The example above, could therefore be rewritten as:

```sas
DATA EX13AI;
ARRAY MOZART A B C D E;
DO OVER MOZART;
  IF MOZART=999 THEN MOZART=.;
END;
CARDS;
(data lines)
```

As convenient as this may seem, we still recommend the explicit arrays of version 6.

XIV. One Final Slightly Clever Example

In this example we will transform a data set from the form shown directly below to one that has all measurements for one subject in one observation. The objective here is just the opposite of that done in example VIII. One possible reason for this transformation would be to use the REPEATED statement in PROC GLM. Here is the original data set:

```
SUBJ TIME X Y Z
1 1 5 6 7
1 2 6 7 8
1 3 6 6 6
1 4 3 4 5
1 5 7 8 9
2 1 4 5 6
2 2 4 5 7
2 3 5 7 9
2 4 5 8 8
2 5 8 8 9
```

The transformed data set would look like this:

```
SUBJ TIME X Y Z
1 1 5 6 7
1 2 6 7 8
1 3 6 6 6
1 4 3 4 5
1 5 7 8 9
2 1 4 5 6
2 2 4 5 7
2 3 5 7 9
2 4 5 8 8
2 5 8 8 9
```

The program which does the transformation, uses an array for each of the variables X, Y, and Z and the RETAIN function to prevent the variables from being set to missing each time a new observation from the original data set is read.

```sas
DATA MULTIPLE;
  INPUT SUBJ TIME X Y Z;
  CARDS;
  1 1 5 6 7
  1 2 6 7 8
  DATA ONEPER;
  SET MULTIPLE;
  BY SUBJ TIME;
  ARRAY XX[*] X1-X5;
  ARRAY YY[*] Y1-Y5;
  ARRAY ZZ[*] Z1-Z5;
  RETAIN X1-X5 Y1-Y5 Z1-Z5;
  IF FIRST.SUBJ THEN DO I=1 TO 5;
    XX[I]=.; YY[I]=.; ZZ[I]=.;
  END;
  XX[TIME]=X;
  YY[TIME]=Y;
  ZZ[TIME]=Z;
  IF LAST.SUBJ THEN OUTPUT;
  KEEP SUBJ X1-X5 Y1-Y5 Z1-Z5;
  RUN;
  PROG SORT DATA=MULTIPLE;
  BY SUBJ TIME;
  RUN;
  DATA ONEPER;
  SET MULTIPLE;
  BY SUBJ TIME;
  KEEP SUBJ X1-X5 Y1-Y5 Z1-Z5;
  RUN;
```

Let's follow this program in detail to see how it works. The first observation from data set MULTIPLE has SUBJ=1, TIME=1, X=5, Y=6, and Z=7. FIRST.SUBJ is TRUE so we set the three arrays (and therefore the values of X1-X5, Y1-Y5, and Z1-Z5) to missing. If we did not do this, a subject with a missing observation (i.e. no data taken at TIME n) would be assigned the values from the previous subject for that time. Next, since TIME=1, XX[TIME] is equal to XX[1] which represents the variable X1 which gets set to 5, the value of X. In the same manner, Y1 will be set to 6 and Z1 will be assigned the value 7. Since LAST.SUBJ is not true, we do not output; we return to read the next observation from MULTIPLE. By using the RETAIN statement, the values of X1-X5, Y1-Y5, and Z1-Z5 will not be reset to missing each time a new observation is read. Thus we can imagine filling in the boxes in the diagram above. The internal SAS
variable LAST.SUBJ is TRUE when the last observation for a subject is read. So when LAST.SUBJ is TRUE, we OUTPUT the observation to the new data set. This program will also work if there are missing observations in the original data set.

XV. Conclusions

Yes, we can live without ARRAYS, but a thorough understanding of ARRAYS can substantially reduce the size of a SAS program. We hope that the examples offered here will give you the courage to try using ARRAYS in your next program. Good luck.

<table>
<thead>
<tr>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS 1</td>
<td>90</td>
<td>90</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>CLASS 2</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>CLASS 3</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>60</td>
</tr>
</tbody>
</table>

Table I

<table>
<thead>
<tr>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVE[1]+X[1,1]/5; CLASS=1, TEST=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[1]+X[1,2]/5; TEST=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[1]+X[1,3]/5; TEST=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[1]+X[1,4]/5; TEST=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[1]+X[1,5]/5; TEST=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[2]+X[2,1]/5; CLASS=2, TEST=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[2]+X[2,2]/5; TEST=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[2]+X[2,3]/5; TEST=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[2]+X[2,4]/5; TEST=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[2]+X[2,5]/5; TEST=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[3]+X[3,1]/5; CLASS=3, TEST=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[3]+X[3,2]/5; TEST=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[3]+X[3,3]/5; TEST=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[3]+X[3,4]/5; TEST=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVE[3]+X[3,5]/5; TEST=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table II