6

Array Processing

6.1 Introduction to Array Processing

If you are familiar with other programming languages, you have probably worked with arrays. Arrays allow you to modify data more efficiently because you can use them to perform the same tasks for a group of related variables inside a looping structure.

Working with arrays, however, is more complicated in SAS®, partly because the syntax has been enriched to allow for nuanced data processing. Even when the syntax is mastered, one still needs to understand how the array interacts with what is going on in the PDV. Knowing when it is best to replace existing code with an array presents an additional challenge. In this chapter, a wide range of applications that use arrays in looping structures will be covered. When you finish reading this chapter, you should know how arrays work and when best to use them in your own SAS programs. The material in this chapter is based upon a paper that I presented at SAS Global Forum (Li, 2011).

6.1.1 Situations for Utilizing Array Processing

The following example illustrates a typical situation for utilizing array processing. The data set SBP contains six measurements of systolic blood pressure (SBP) measurements for four patients. The missing values are coded as 999. Suppose that you would like to recode 999 to the SAS value for missing (.). You may want to write the code like the one in Program 6.1.

SBP:

	SBP1	SBP2	SBP3	SBP4	SBP5	SBP6
1	141	142	137	117	116	124
2	999	141	138	119	119	122
3	142	999	139	119	120	999
4	136	140	142	118	121	123

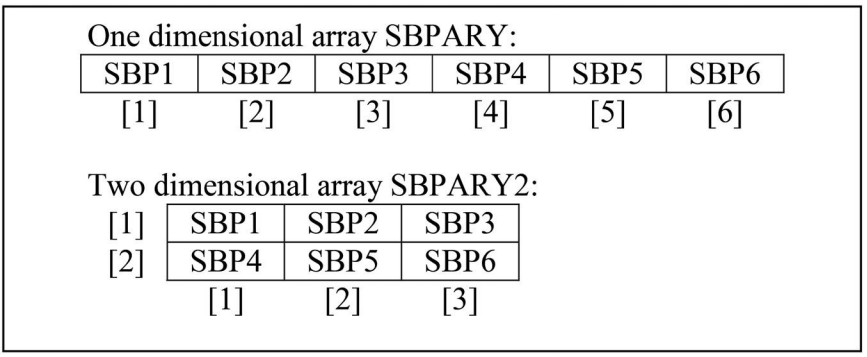

FIGURE 6.1
The conceptual view of one- and two-dimensional arrays.

Program 6.1:

```
data ex6_1;
    set sbp;
    if sbp1 = 999 then sbp1 = .;
    if sbp2 = 999 then sbp2 = .;
    if sbp3 = 999 then sbp3 = .;
    if sbp4 = 999 then sbp4 = .;
    if sbp5 = 999 then sbp5 = .;
    if sbp6 = 999 then sbp6 = .;
run;
```

Each of the IF statements in Program 6.1 converts the number 999 to a missing SAS value. These IF statements are almost identical; only the names of the variables are different. If these six variables can be grouped into a one-dimensional array (see Figure 6.1), then you can recode the variables in a DO loop. In this situation, grouping variables into an array will make code writing more efficient.

In Figure 6.1, variables SBP1 to SBP6 are grouped into a one-dimensional array, SBPARY. Once these variables are grouped into an array, you can reference the data in these variables by using the SBPARY[*n*] format, where *n* is the element number. For example, to reference the data in the SBP3 variable, you can use SBPARY[3].

Variables in the DATA step can also be grouped into multi-dimensional arrays. For example, in the bottom portion of Figure 6.1, SBP1 to SBP6 are grouped into a two-dimensional array, SBPARY2. To reference the data in the variables in SBPARY2, you need to use the SBPARY2[*r*, *c*] format, where *r* is the row number and *c* is the column number. For example, to reference the data in the SBP3 variable, you can use SBPARY2[1,3]. To reference the data in SBP5, use SBPARY2[2,2].

6.1.2 Defining and Referencing One-Dimensional Arrays

A SAS array is a temporary grouping of SAS variables under a single name. Arrays exist only for the duration of the DATA step. The ARRAY statement is used to either group previously defined variables or to create a group of variables, which has the following form:

```
ARRAY array-name [subscript] <$><length><array-elements>
      <(initial-value-list)>;
```

The array-name in the ARRAY statement does not become part of the output data. The *array-name* must be a legitimate SAS name and cannot be the name of a SAS variable in the same DATA step. If you happen to use a function name as *array-name*, this name would be treated as the name of an array in parenthetical references, not the name of the function. SAS will also send a warning message to your SAS log. Furthermore, *array-name* cannot be used in the LABEL, FORMAT, DROP, KEEP, or LENGTH statements.

The [subscript] component in the ARRAY statement describes the number or arrangement of array elements and can be specified in different forms. You can enclose *subscript* with braces {}, brackets [], or parentheses (). The simple form for [subscript] is to specify the dimensional size of the array. For example, you can group SBP1 to SBP6 into an array like below:

```
array sbpary[6] sbp1 sbp2 sbp3 sbp4 sbp5 sbp6;
```

You can also provide a range of numbers as [subscript] by providing the lower and upper bounds of the array and separate them by a colon (:). For example,

```
array sbpary[1990:1993] sbp1990 sbp1991 sbp1992 sbp1993;
```

An asterisk (*) can also be used as [subscript]. Using an asterisk will let SAS determine the subscript by counting the variables in the array. When you specify the asterisk, you must include *array-elements*. For example,

```
array sbpary[*] sbp1 sbp2 sbp3 sbp4 sbp5 sbp6;
```

The optional $ (dollar sign) indicates that the elements in the array are character elements. You don't need to specify the $ if the array elements have been previously defined as character elements. If the lengths of array elements have not been previously specified, you can use the *length* option in the ARRAY statement.

The optional *array-elements* are the variables to be included in the array, which must either be all numeric or character variables. If variable names that are included in the array end with consecutive numeric values, you can use a shorthand notation by connecting the first and last variables with a hyphen. For example, SBP1-SBP6 is equivalent to listing SBP1 to SBP6 individually.

If *array-elements* are not specified, the *array-elements* would be implied to the variables with the names that contain array names and the numbers 1, 2, … n. For example,

```
array sbp[6];
```

is equivalent to the following statement:

```
array sbp[6] sbp1-sbp6;
```

When *array-elements* (SBP1-SBP6) are not listed, SBP1 to SBP6 can already exist in the DATA step. However, if SBP1 to SBP6 do not already exist, they will be created in the DATA step.

Another way to list *array-elements* is to use the keywords _NUMERIC_, _CHARACTER_, and _ALL_, which are used to specify all numeric, all character, or all the same type of variables. If the keyword _ALL_ is used, all the previously defined variables must have the same type. For example,

```
array num[*] _numeric_;
array char[*] _character_;
array allvar[*] _all_;
```

You can use the keyword _TEMPORARY_ as *array-elements* to create temporary arrays. Using temporary arrays is useful when you want to create an array only for computing purposes. When referring to a temporary data element, you refer to it by the *array-name* and its dimension. Because the temporary array contains only constants as elements, they cannot be sent as variables to the output data set. Also, the values in temporary arrays are automatically retained without being reset to missing at the beginning of each iteration of the DATA step execution. You cannot use asterisks (*) with temporary arrays.

You can also assign initial values to the array elements in the (*initial-value-list*) when creating a group of variables by using the ARRAY statement. The initial values can also be assigned to temporary data elements. The values can be either numbers or character strings separated by either a comma or a blank space. When any or all elements in an array are assigned with initial values, all elements in the array would act as they are named in a RETAIN statement. For example, the following ARRAY statement creates N1, N2, and N3 DATA step variables by using the ARRAY statement and initializes them with the values 1, 2, and 3, respectively:

```
array num[3] n1 n2 n3 (1 2 3);
```

The following ARRAY statement creates CHR1, CHR2, and CHR3 variables. The dollar sign ($) is necessary because CHR1, CHR2, and CHR3 are not previously defined in the DATA step.

```
array chr[3] $ ('A', 'B', 'C');
```

Next, the ARRAY statement creates a temporary array, NUM, and the number of elements in the array is 3. Each element in the array is initialized to 1, 2, and 3:

```
array num[3] _temporary_ (1 2 3);
```

Again, notice that NUM1, NUM2, and NUM3 are not created as variables, because a temporary array is being defined.

After an array is defined, you can use the ARRAY reference statement to describe the elements in an array to be processed in the DATA step. The ARRAY reference statement has the following form:

```
array-name[subscript]
```

The *array-name* is the name of an array that was previously defined within the sample DATA step. The [*subscript*] component is used to specify the subscript of the array. You can use the name of a variable, an asterisk (*), or a SAS expression as the *subscript*.

When a variable is used as the *subscript*, the array referencing it is used with an iterative DO loop. Within each iteration of execution of the DO loop, the current value of this variable is used as the *subscript* of the array element being processed in the DATA step.

When an asterisk (*) is used as the *subscript*, SAS will treat the elements in the array as a variable list. You cannot reference the temporary array elements with an asterisk.

When a SAS expression is used as *subscript* in a SAS statement, the expression needs to be evaluated to a value when the statement executes. The evaluated value needs to be within the lower and upper bounds of the array. You can also use an integer value as *subscript.* Program 6.2 is a modified version of Program 6.1 by using array processing.

Program 6.2:

```
data ex6_2(drop = i);
    set sbp;
    array sbpary[6] sbp1-sbp6;
    do i = 1 to 6;
        if sbpary[i] = 999 then sbpary[i] = .;
    end;
run;
```

6.1.3 Compilation and Execution Phases for Array Processing

During the compilation phase, the PDV is created (_ERROR_ is omitted for simplicity). The array name SBPARY and array references are not included in the PDV. (See Figure 6.2.) Each variable, SBP1 to SBP6, is referenced by

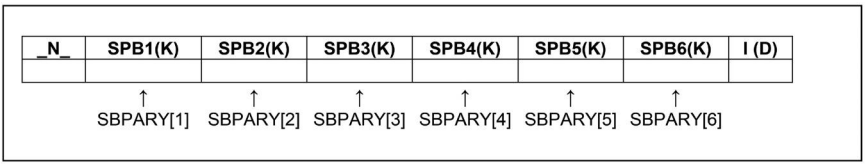

FIGURE 6.2
The array references are not included in the program data vector (PDV) during the compilation phase.

the array reference. Syntax errors in the ARRAY statement will be detected during the compilation phase.

The first two iterations of the DATA step execution are illustrated in Figures 6.3 and 6.4.

6.2 Functions and Operators Related to Array Processing

6.2.1 The DIM, HBOUND, and LBOUND Functions

Sometimes you won't know the dimension of an array. This tends to be the case when you use _NUMERIC_, _CHARACTER_, and _ALL_ as *array-elements*. A handy work-around is the DIM function, which supplies the actual dimension to the DATA step. Closely related to the DIM function are the HBOUND and LBOUND functions that return the upper and lower bounds of an array, respectively. The three functions have the following forms:

```
DIM<n>(array-name)
DIM(array-name,bound-n)
HBOUND<n>(array-name)
HBOUND(array-name,bound-n)
LBOUND<n>(array-name)
LBOUND(array-name,bound-n)
```

Notice that these functions share similar syntax. The optional *n* is used to specify the dimension of an array. If the *n* value is not specified, the DIM function will return the number of elements in the first dimension of the array, and HBOUND and LBOUND will return the upper bound and lower bound of the first dimension, respectively. The *bound-n* is a numeric constant, variable, or SAS expression used to specify the dimension for which you want to know the number of elements in a multi-dimensional array from the DIM function. If *bound-n* is used in

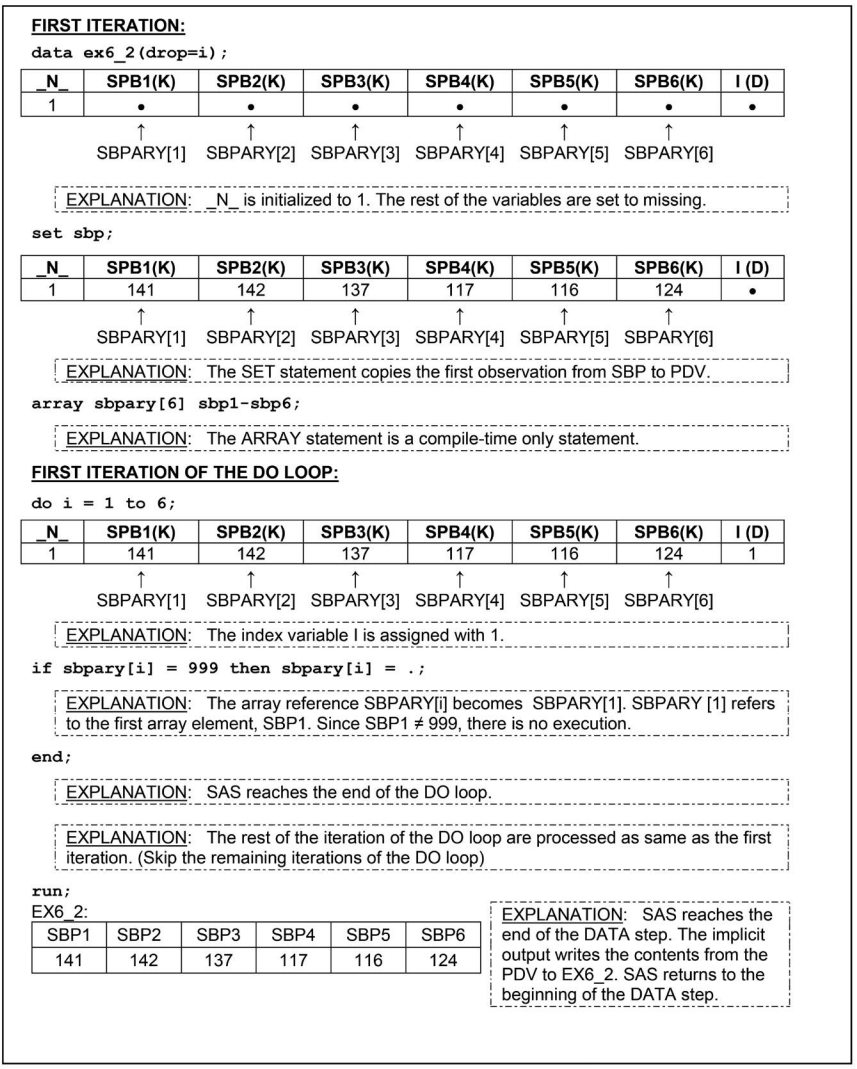

FIGURE 6.3
The first iteration of the DATA step in Program 6.2.

the HBOUND or LBOUND functions, the upper or lower bound of the array will be returned. You can only use *bound-n* when *n* is not specified. For example, in Program 6.3 the DIM function is used to return the number of elements in SBPARY. Instead of using the DIM function, you can also use the HBOUND function to return the upper bound of SBPARY, which is 6.

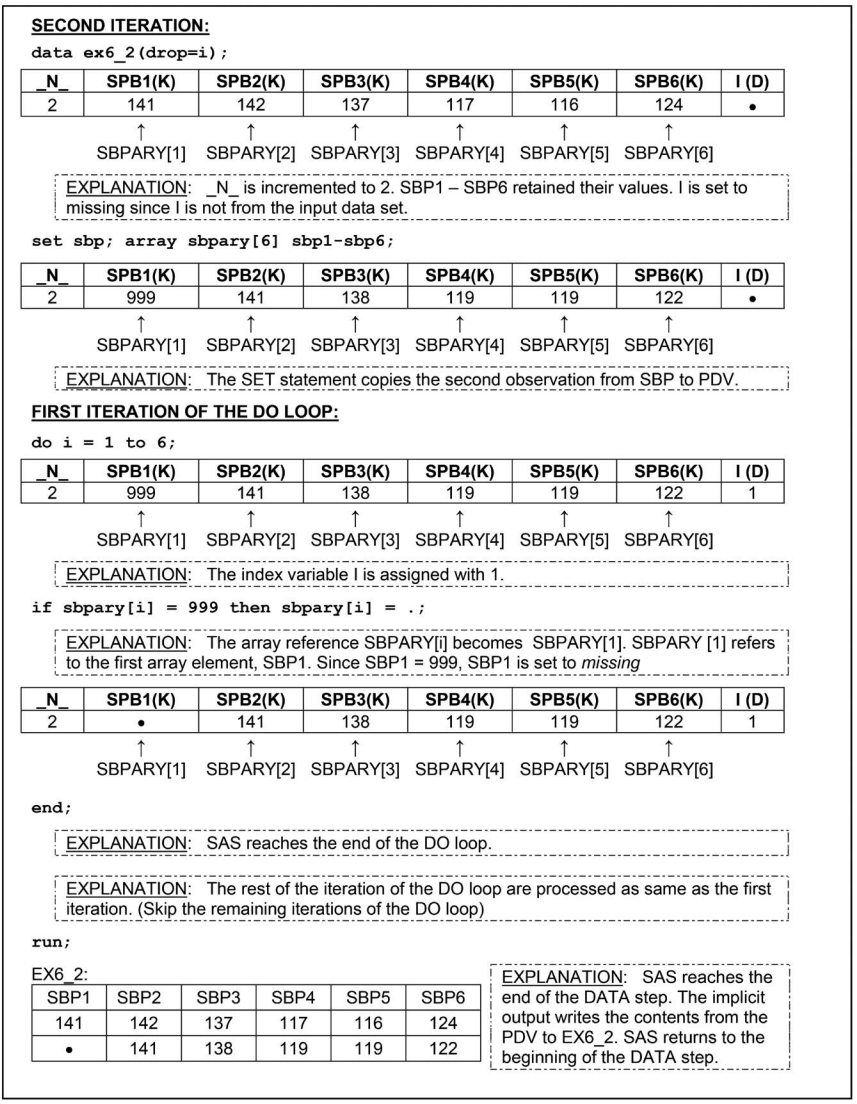

FIGURE 6.4
The second iteration of the DATA step in Program 6.2.

Program 6.3:

```
data ex6_3(drop = i);
    set sbp;
    array sbpary[*] _numeric_;
    do i = 1 to dim(sbpary); /*Or: do i = 1 to hbound(sbpary)*/
```

```
        if sbpary[i] = 999 then sbpary[i] =.;
    end;
run;
```

LBOUND and HBOUND are typically used when the lower bound of an array dimension has a value other than 1 or the upper bound has a value other than *n*.

6.2.2 Using the IN and OF Operator with an Array

The data set SBP2 is similar to the data set SBP except that SBP2 contains the data with the correct numerical missing values:

SBP2:

	SBP1	SBP2	SBP3	SBP4	SBP5	SBP6
1	141	142	137	117	116	124
2	.	141	138	119	119	122
3	142	.	139	119	120	.
4	136	140	142	118	121	123

For example, suppose that you would like to create a variable, MISS, that is used to indicate whether SBP1–SBP6 contains missing values. This task can be easily accomplished by using the IN operator.

The IN operator, introduced in Chapter 1, is used to determine whether a variable's value is among the list of character or numeric values. You can use the IN operator to search for numeric or character values in an array. Program 6.4 illustrates the use of the IN operator with SBPARY to create the MISS variable.

Program 6.4:

```
data ex6_4;
    set sbp2;
    array sbpary[*] _numeric_;
    miss = . IN sbpary;
run;

title 'Using the IN operator to create variable MISS';
proc print data = ex6_4;
run;
```

Output from Program 6.4:

```
   Using the IN operator to create variable MISS
 Obs   sbp1  sbp2   sbp3   sbp4    sbp5   sbp6   miss
  1    141   142    137    117     116    124     0
```

```
  2     .    141    138    119     119    122     1
  3    142    .     139    119     120     .      1
  4    136   140    142    118     121    123     0
```

You can pass an array on to most functions with the OF operator. Suppose that you would like to create two variables, SBP_MIN and SBP_MAX, that contain the minimum and maximum SBP values for each person. You can use the MIN and MAX function with the OF operator to accomplish this task. (See Problem 6.5.) Note that you must use an asterisk as the subscript within the function.

Program 6.5:

```
data ex6_5;
    set sbp2;
    array sbpary[*] _numeric_;
    sbp_min = min(of sbpary[*]);
    sbp_max = max(of sbpary[*]);
run;

title 'Using the OF operator to create variables SBP_MIN and
SBP_MAX';
proc print data = ex6_5;
run;
```

Output from Program 6.5:

```
  Using the OF operator to create variables SBP_MIN and SBP_MAX
 Obs  sbp1  sbp2  sbp3  sbp4  sbp5  sbp6  sbp_min  sbp_max
  1    141   142   137   117   116   124    116      142
  2     .    141   138   119   119   122    119      141
  3    142    .    139   119   120    .     119      142
  4    136   140   142   118   121   123    118      142
```

6.3 Some Array Applications

6.3.1 Creating a Group of Variables by Using Arrays

Suppose that the first three measurements of SBP in the SBP2 data set are recorded pre-treatment and the last three are recorded post-treatment. Furthermore, suppose that the average SBP value for the pre-treatment measurements is 140 and that the average SBP is 120 for the measurements after the treatments. You would like to create a list of variables, ABOVE1–ABOVE6, that indicate whether each measurement is above (1) or below (0) the average measurement.

The solution for this problem is in Program 6.6. The first ARRAY statement in Program 6.6 is used to group the existing variables, SBP1–SBP6. The second

ARRAY statement creates six new DATA step variables, ABOVE1 through ABOVE6. The third ARRAY statement creates temporary data elements used only for comparison purposes.

Program 6.6:

```
data ex6_6(drop = i);
    set sbp2;
    array sbp[6];
    array above[6];
    array threshold[6] _temporary_ (140 140 140 120 120 120);
    do i = 1 to 6;
        if (not missing(sbp[i]))
            then above[i] = sbp[i] > threshold[i];
    end;
run;

title 'Creating variables ABOVE1 - ABOVE6';
proc print data = ex6_6;
run;
```

Output from Program 6.6:

```
       Creating variables ABOVE1 - ABOVE6
                                          a   a   a   a   a   a
                                          b   b   b   b   b   b
      s     s     s     s     s     s     o   o   o   o   o   o
 O    b     b     b     b     b     b     v   v   v   v   v   v
 b    p     p     p     p     p     p     e   e   e   e   e   e
 s    1     2     3     4     5     6     1   2   3   4   5   6
 1   141   142   137   117   116   124     1   1   0   0   0   1
 2    .    141   138   119   119   122     .   1   0   0   0   1
 3   142    .    139   119   120    .      1   .   0   0   0   .
 4   136   140   142   118   121   123     0   0   1   0   1   1
```

6.3.2 Calculating Products of Multiple Variables

You can use the SUM function to calculate the sum of multiple variables. However, SAS does not have a built-in function to calculate the *product* of multiple variables. The easiest way to calculate the product of multiple variables is to use array processing. For example, in Program 6.7, the product of NUM1–NUM4 is calculated for each observation in the PRODUCT data set with array processing.

PRODUCT:

	NUM1	NUM2	NUM3	NUM4
1	4	.	2	3
2	.	2	3	1

Program 6.7:

```
data ex6_7(drop = i);
    set product;
    array num[4];
    if missing(num[1]) then result = 1;
    else result = num[1];
    do i = 2 to 4;
        if not missing(num[i]) then result = result*num[i];
    end;
run;

title 'Calculating the product of NUM1-NUM4';
proc print data = ex6_7;
run;
```

Output from Program 6.7:

Calculating the product of NUM1-NUM4

Obs	num1	num2	num3	num4	result
1	4	.	2	3	24
2	.	2	3	1	6

6.3.3 Restructuring Data Sets Using One-Dimensional Arrays

Transforming a data set with one observation per subject to multiple observations per subject, or vice versa, was introduced in Sections 3.3.3 and 4.2.4. The solutions in these two sections are not efficient if you have large numbers of variables that need to be transposed. A more effective approach is to utilize array processing. The next two programs use the same data sets, WIDE and LONG, that were introduced in Chapter 3.

In Program 6.8, the variables S1–S3 are grouped into the array S, and the program utilizes the iterative DO loop with the TIME variable as the index variable (to assign S[TIME] to the SCORE variable). The output is generated within the DO loop when the SCORE variable is not missing. Program 6.8 transforms data set WIDE into a *long* format with array processing.

Program 6.8:

```
data ex6_8(drop = s1-s3);
    set wide;
    array s[3];
    do time = 1 to 3;
        score = s[time];
        if not missing(score) then output;
    end;
run;
```

Program 6.9 does the opposite by transforming data set LONG to a *wide* format. Array processing is still used in the second transformation. Variables S1, S2, and S3 are created by using the ARRAY statement. You can retain all the elements in the array by using the RETAIN statement and the name of the array. When reading the first observation of each subject (FIRST.ID = 1), an iterative DO loop is used to initialize each element in the array (S[I]) to missing. Utilizing array processing simplifies processing shown in Program 4.5 from Chapter 4 because the cumbersome IF-THEN/ELSE statement is replaced with an assignment statement that uses TIME as the *subscript* of the S array (S[TIME] = SCORE).

Program 6.9:

```
data ex6_9 (drop = time score i);
    set long;
    by id;
    array s[3];
    retain s;
    if first.id then do;
        do i = 1 to 3;
            s[i] = .;
        end;
    end;
    s[time] = score;
    if last.id;
run;
```

6.4 Applications That Use Multi-Dimensional Arrays

The syntax for creating multi-dimensional arrays is similar to the one for creating one-dimensional arrays. The only difference is that multiple numbers or ranges of numbers appear in the array *subscript*. The numbers for each dimension are separated by a comma. The first number of the *subscript* refers to the number of rows, and the second number refers to the number of columns. If there are three dimensions, the next number will refer to the number of pages.

6.4.1 Calculating Average SBP for Pre- and Post-Treatment

Suppose that the SBP2 data set contains three pre-treatment measurements of SBP (SBP1–SBP3) and three post-treatment measurements of SBP values (SBP4–SBP6). You would like to create two variables that contain average SBP for pre- and post-treatments for each patient. One way to solve this problem is to use a two-dimensional array as in Program 6.10.

Program 6.10:

```
data ex6_10 (drop = i j);
    set sbp2;
    array sbp[2, 3];  /* Row 1 = Pre Treatment, Row 2 = Post
                      Treatment */
                      /* Columns 1-3 are SBP measurements */
    array sbpmean[2]; /* Two means per subject */
    array n[2] _temporary_;
    array sbpsum [2] _temporary_;
    do i = 1 to 2;
        sbpsum[i] = 0;
        n[i] = 0;
        do j = 1 to 3;
            sbpsum[i] + sbp[i,j];
            if not missing(sbp[i,j]) then n[i] + 1;
        end;
        sbpmean[i] = sbpsum[i]/n[i];
    end;
run;

title 'Calculating the average of SBP for pre- and post-treat-
ment';
proc print data = ex6_10;
run;
```

Output from Program 6.10:

Calculating the average of SBP for pre- and post-treatment

Obs	sbp1	sbp2	sbp3	sbp4	sbp5	sbp6	sbpmean1	sbpmean2
1	141	142	137	117	116	124	140.000	119.000
2	.	141	138	119	119	122	139.500	120.000
3	142	.	139	119	120	.	140.500	119.500
4	136	140	142	118	121	123	139.333	120.667

Program 6.10 uses a 2-by-3 two-dimensional array to group SBP1–SBP6; the first row of the array will contain the pre-treatment measurements (SBP1–SBP3), and the second row will contain the post-treatment results (SBP4–SBP6). There will be two variables that contain the mean measures for pre- and post- measurements; thus, a one-dimensional array (SBPMEAN) with two elements can be used to hold these two values for each patient. A one-dimensional array (N) is used to accumulate the non-missing measurements used to calculate the mean. A nested loop is used in the program. There are two iterations for the outer loop: one for pre-treatment measurements and one for post-treatment measurements. There are three iterations for the inner loop because there are three measurements for the pre- and post-measurements.

6.4.2 Restructuring Data Sets by Using a Multi-Dimensional Array

The DAT1 data set in this section contains two records for each person, whereas the DAT2 data set contains the same information as DAT1 (except that DAT2 contains one record for each person). To transform DAT1 to DAT2 (or vice versa), you will need to use a two-dimensional array.

DAT1:

	ID	G1	G2	G3
1	1	A	B	F
2	1	B	A	C
3	2	B	A	D
4	2	C	B	C

DAT2:

	ID	M_G1	M_G2	M_G3	F_G1	F_G2	F_G3
1	1	A	B	F	B	A	C
2	2	B	A	D	C	B	C

Program 6.11 transforms data set DAT1 to data set DAT2. Because you are only creating the observation after you finish reading all the observations for each person, you need to use ID as the BY variable. Output is generated when LAST.ID equals 1. A one-dimensional array, G[3], is used to group the existing variables (G1–G3) from the input data set. A two-dimensional array, ALL_G[2,3], is used to create variables M_G1, M_G2, M_G3, F_G1, F_G2, and F_G3. The first dimension for ALL_G[2,3] is 2, which corresponds to the number of observations for each subject. The second dimension for ALL_G[2,3] is 3, which corresponds to the number of variables (G1–G3) that need to be transformed from the input data set. The SUM statement is used to increment index I in the implicit loop of the DATA step. Also, the iterative DO loop starts over with J = 1 when the implicit loop of the DATA step is executed. The newly created variables in the ALL_G array need to be RETAINED because to completely fill the array requires two (not one) iterations of the implicit loop of the DATA step. Otherwise the first three values in the array would always be reset to missing when all six are written out in the second iteration when LAST.ID is true (1). (*Program 6.11* is also illustrated in *program6.11.pdf*.)

Program 6.11:

```
proc sort data = dat1;
    by id;
run;

data dat2(drop = i j g1 - g3);
    set dat1;
```

```
    by id;
    array all_g[2,3] $ m_g1 - m_g3 f_g1 - f_g3;
    array g[3];
    retain all_g;
    if first.id then i = 0;
    i + 1;
    do j = 1 to 3;
        all_g[i,j] = g[j];
    end;
    if last.id;
run;
```

Program 6.12 transforms data set DAT2 back to data set DAT1. The thinking behind Program 6.12 is similar to Program 6.11. A one-dimensional array, G[3], is used to create variables G1–G3. A two-dimensional array, ALL_G[2,3], is used to group the existing variables M_G1, M_G2, M_G3, F_G1, F_G2, and F_G3. The nested loop is used to create variables G1–G3. The OUTPUT statement that follows the inner loop is needed to create two observations per iteration of the outer loop.

Program 6.12:

```
data dat1(drop = i j m_g1 - f_g3);
    set dat2;
    array all_g[2,3] m_g1 - f_g3;
    array g[3] $;
    do i = 1 to 2;
      do j = 1 to 3;
         g[j] = all_g[i,j];
      end;
      output;/* After the inner J loop and before the end of the
                outer I loop */
    end;
run;
```


Exercises

Exercise 6.1. Consider the following data set, PROB6_1.SAS7BDAT:

	ID	G1	G2	G3	S1	S2	S3
1	1	A	A	C	3	4	9
2	2	A	B	F	3	7	4
3	3	A	C	B	5	8	9

Transform PROB6_1 to multiple observations per subject by using array processing like below:

	ID	TIME	GRADE	SCORE
1	1	1	A	3
2	1	2	A	4
3	1	3	C	9
4	2	1	A	3
5	2	2	B	7
6	2	3	F	4
7	3	1	A	5
8	3	2	C	8
9	3	3	B	9

Exercise 6.2. The PROB6_2.sas7bdat data set contains answers for a multiple-choice test for 1,000 students. There are 100 questions for this test (Q1–Q100). In addition to the Q1–Q100 variables, there is an ID variable in this data set. The first observation is the solution of the test. The next 1,000 observations are the answers from the students. Based on this data set, create a variable, SCORE, that is the number of correct answers for each student. For example, if student S1 has 82 correct answers compared to the KEY value, the SCORE will equal 82 for S1.

Exercise 6.3. Perry Watts's candidate exercise. Program 6.6 creates ABOVE1–ABOVE6 by comparing whether each measurement of SBP from the SBP2 data set is above (1) or below (0) the average SBP measurement of pre-treatment (140) and post-treatment measurements (120). Write a program by using array processing to calculate the average SBP values for pre-treatment (SBP1–SBP3) and post-treatment (SBP4–SBP 6) values for the four patients in the SBP2 data set.