Paper 244-2011

The Many Ways to Effectively Utilize Array Processing

Arthur Li, City of Hope Comprehensive Cancer Center, Duarte, CA

ABSTRACT

Utilizing array processing allows you to reduce the amount of coding in the DATA step. Some may find that using array processing is confusing or even intimidating. First of all, the syntax for constructing arrays, especially for multi-dimensional arrays, is complicated; there are way too many rules. Once mastering syntax rules, you still need to understand what happens in the Program Data Vector (PDV) during array processing; otherwise, the data set that you created by using array processing may not be the one that you intended. Lastly, knowing when best to use array processing is another challenge.

In this section, in addition to learning how to create one- and multi-dimensional arrays, you will also receive a brief review on how to create an explicit loop in the DATA step - the prerequisite for constructing an array. You will also be exposed to what happens in the PDV during array processing. A wide range of applications in using loop structures with array processing will also be covered in this talk as well.

INTRODUCTION

If you've known other programming languages, you are probably familiar with an array, which is a type of data structure. However, an array in SAS® is completely different, which is used to temporarily group a set of variables. Grouping variables by using an array allows you to modify data more efficiently since you will be able to utilize loops to work along the grouped variables.

For example, the following dataset contains six measurements of systolic blood pressure (SBP) for four patients. The missing values are coded as 999. Suppose that you would like to recode 999 to periods (.). You may want to write the following code like the one in Program 1 below.

Sbp.sas7bdat:

	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

Program 1:
```
data sbp1;
    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 1* changes 999 to a missing value. These IF statements are almost identical; only the name of the variables are different. If we can group these six variables into a one-dimensional array, then we can recode the variables in a DO loop. In this situation, grouping variables into an array will make code writing more efficient.

REVIEW OF IMPLICIT AND EXPLICIT LOOPS

Since array processing is closely related to the explicit loop in the DATA step, a brief review of loop structure is essential to understanding array processing. In SAS, there are implicit and explicit loops. SAS programmers often find differences between implicit and explicit loops confusing. The implicit loop refers to the iteration of the DATA step, which is related to the DATA step execution. Thus, in order to understand the loop structure, we also need to review the compilation and execution of the DATA step.

COMPILATION AND EXECUTION PHASES

A DATA step is processed in a two-phase sequence: compilation and execution phases. In the compilation phase, each statement is scanned for syntax errors. The PDV[1] is created according to the descriptor portion of the input dataset.

The execution phase begins after the compilation phase. In the execution phase, SAS uses the PDV to build the new dataset. During the execution phase, the DATA step works like a loop, repetitively reading data values from the input dataset, executing statements, and creating observations for the output dataset one at a time. This is the implicit loop. SAS stops reading the input file when it reaches the end-of-file marker, which is located at the end of the input data file. At this point, the implicit loop ends.

IMPLICIT LOOPS

The following example shows how the implicit loop is processed. Suppose that you would like to assign each subject in a group of patients with a drug or a placebo. The chance of receiving the actual drug is 50%. For illustration purposes, only four patients from the trial are used in the example. The dataset is similar to the one below. The solution is shown in *program 2*.

Patient[2]

	ID
1	M2390
2	F2390
3	F2340
4	M1240

Program 2:
```
data example1 (drop=rannum);
    set patient;
    rannum = ranuni(2);
    if rannum> 0.5 then group = 'D';
    else group ='P';

proc print data=example1;
run;
```

Output:
```
 Obs      ID       group

  1      M2390       P
  2      F2390       D
  3      F2340       D
  4      M1240       D
```

At the beginning of the execution phase, the automatic variable _N_ is initialized to 1 and _ERROR_ is initialized to 0 inside the PDV. _N_ is used to indicate the current observation number. _ERROR_ is more often used to signal the data entry error. The non-automatic variables are set to missing.

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0		•	

Next, the SET statement copies the first observation from the dataset *patient* to the PDV. The variable RANNUM is generated by the **RANUNI**[3] function. Since RANNUM is not greater than 0.5, GROUP is assigned with value ‘P’.

[1] Program Data Vector is a memory area on your computer to build the new dataset
[2] A SAS data set ends with *.sas7bdat. For simplicity purposes, the file extension is omitted in this paper.
[3] The **RANUNI** function generates a number following uniform distribution between 0 and 1. The general form is **RANUNI(seed)**, where seed is a nonnegative integer. The RANUNI function generates a stream of numbers based on *seed*. When *seed* is set to 0, which is the computer clock, the generated number cannot be reproduced. However, when *seed* is a non-zero number, the generated number can be produced.

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	M2390	0.3699251396	P

The implicit OUTPUT statement at the end of the DATA step tells SAS to write the contents from the PDV to the dataset *example1*. The SAS system returns to the beginning of the DATA step to begin the second iteration.

At the beginning of the second iteration, since data is read from an existing SAS dataset, the value in the PDV for the ID variable is retained from the previous iteration. The newly-created variables RANNUM and GROUP are initialized to missing[4]. The automatic variable _N_ is incremented to 2. Next, RANNUM is generated and GROUP is assigned to 'D'. The implicit OUTPUT statement tells SAS to write the contents from the PDV to the output dataset *example1*. The SAS system returns to the beginning of the DATA step to begin the third iteration.

The entire process for the third and fourth iterations is similar to the previous iterations. Once the fourth iteration is completed, SAS returns to the beginning of the DATA step again. At this time, when SAS attempts to read an observation from the input dataset, it reaches the end-of-file-marker, which means that there are no more observations to read. Thus, the execution phase is completed.

EXPLICIT LOOP
In the previous example, the patient ID is stored in an input dataset. Suppose you don't have a dataset containing the patient IDs. You are asked to assign four patients with a 50% chance of receiving either the drug or the placebo. Instead of creating an input dataset that stores ID, you can create the ID and assign each ID to a group in the DATA step at the same time. For example,

Program 3:
```
data example2(drop = rannum);
    id = 'M2390';
    rannum = ranuni(2);
    if rannum> 0.5 then group = 'D';
    else group ='P';
    output;

    id = 'F2390';
    rannum = ranuni(2);
    if rannum> 0.5 then group = 'D';
    else group ='P';
    output;

    id = 'F2340';
    rannum = ranuni(2);
    if rannum> 0.5 then group = 'D';
    else group ='P';
    output;

    id = 'M1240';
    rannum = ranuni(2);
    if rannum> 0.5 then group = 'D';
    else group ='P';
    output;
run;
```

[4] When creating a SAS dataset based on a raw dataset, SAS sets each variable value in the PDV to *missing* at the beginning of each iteration of execution, except for the automatic variables, variables that are named in the RETAIN statement, variables created by the SUM statement, data elements in a _TEMPORARY_ array, and variables created in the options of the FILE/INFILE statement. When creating a SAS dataset based on a SAS dataset, SAS sets each variable to missing in the PDV *only* before the first iteration of the execution. Variables will retain their values in the PDV until they are replaced by the new values from the input dataset. These variables exist in both the input and output datasets. However, the newly-created variable, which only exists in the output dataset, will be set to *missing* in the PDV at the beginning of *every* iteration of the execution.

The DATA step in *Program 3* begins with assigning ID numbers and then assigns the group value based on a generated random number. There are four explicit OUTPUT statements[5] that tell SAS to write the current observation from the PDV to the SAS dataset immediately, not at the end of the DATA step. This is what we intended to do. However, without using the explicit OUTPUT statement, we will only create one observation for ID =M1240. Notice that most of the statements above are identical. To reduce the amount of coding, you can simply rewrite the program by placing repetitive statements in a DO loop. Following is the general form for an iterative DO loop:

DO INDEX-VARIABLE = VALUE1, VALUE2, …, VALUEN***;***
SAS STATEMENTS
END;

In the iterative DO loop, you must specify an INDEX-VARIABLE that contains the value of the current iteration. The loop will execute along VALUE1 through VALUEN and the VALUES can be either character or numeric. Here's the improved version of *Program 3*:

Program 4:
```
data example3 (drop = rannum);
    do id = 'M2390', 'F2390', 'F2340', 'M1240';
        rannum = ranuni(2);
        if rannum> 0.5 then group = 'D';
        else group ='P';
        output;
    end;

proc print data=example3;
run;
```

Output:
```
 Obs     id      group

  1     M2390      P
  2     F2390      D
  3     F2340      D
  4     M1240      D
```

THE ITERATIVE DO LOOP ALONG A SEQUENCE OF INTEGERS
More often the iterative DO loop along a sequence of integers is used.

DO INDEX-VARIABLE = START **TO** STOP <**BY** INCREMENT>***;***
SAS STATEMENTS
END;

The loop will execute from the START value to the END value. The optional BY clause specifies an increment between START and END. The default value for the INCREMENT is 1. START, STOP, and INCREMENT can be numbers, variables, or SAS expressions. These values are set upon entry into the DO loop and cannot be modified during the processing of the DO loop. However, the INDEX-VARIABLE can be changed within the loop.

Suppose that you are using a sequence of numbers, say 1 to 4, as patient IDs; you can rewrite the previous program as below:

[5] By default, every DATA step contains an implicit OUTPUT statement at the end of the DATA step that tells the SAS system to write observations to the dataset. Placing an explicit OUTPUT statement in a DATA step overrides the implicit output; in other words, the SAS system adds an observation to a dataset only when an explicit OUTPUT statement is executed. Once an explicit OUTPUT statement is used to write an observation to a dataset, there is no longer an implicit OUTPUT statement at the end of the DATA step.

Program 5:
```
data example4 (drop = rannum);
    do id = 1 to 4;
        rannum = ranuni(2);
        if rannum> 0.5 then group = 'D';
        else group ='P';
        output;
    end;

proc print data=example4;
run;
```

Output:
```
 Obs     id      group


   1      1      P
   2      2      D
   3      3      D
   4      4      D
```

CREATING A ONE-DIMENSIONAL ARRAY

ARRAY DEFINITION AND SYNTAX

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

ARRAY ARRAYNAME[DIMENSION] <$> <ELEMENTS>;

ARRAYNAME in the ARRAY statement must be a SAS name that is not the name of a SAS variable in the same DATA step. DIMENSION is the number of elements in the array. The optional $ sign indicates that the elements in the array are character elements; The $ sign is not necessary if the elements have been previously defined as character elements. ELEMENTS are the variables to be included in the array, which must either be all numeric or characters. For example, you can group variables SBP1 to SBP6 into an array like below:

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

You cannot use ARRAYNAME in the LABEL, FORMAT, DROP, KEEP, or LENGTH statements. Furthermore, ARRAYNAME does not become part of the output data. You should also avoid using the name of the SAS function as an array name because it can cause unpredictable results. For example, if you use a function name as the name of the array, SAS treats parenthetical references that involve the name as array references, not function references, for the duration of the DATA step.

You can specify DIMENSION in different forms. For instance, you can list the range of values of the dimension, like below:

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

You can use an asterisk (*) as DIMENSION. Using an asterisk will let SAS determine the subscript by counting the variables in the array. When you specify the asterisk, you must include ELEMENTS. For example,

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

DIMENSION can be enclosed in parentheses, braces, or brackets. The following three statements are equivalent:

```
array sbparray (6) sbp1 sbp2 sbp3 sbp4 sbp5 sbp6;
```

```
array sbparray {6} sbp1 sbp2 sbp3 sbp4 sbp5 sbp6;
```

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

ELEMENTS in the ARRAY statement is optional. If ELEMENTS is not specified, new DATA step variables will be created with default names. The default names are created by concatenating ARRAYNAME with the array index. For example, `array sbp [6];`

is equivalent to

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

or

`array sbp [6] sbp1 - sbp6;`

The keywords _NUMERIC_, _CHARACTER_, and _ALL_ can all be used to specify all numeric, all character, or all the same type variables.

`array num [*] _numeric_;`

`array char [*] _character_;`

`array allvar [*] _all_;`

After an array is defined, you need to reference any element of an array by using the following syntax:

```
ARRAYNAME[INDEX]
```

When referencing an array element, INDEX must be closed in parentheses, braces, or brackets. INDEX can be specified as an integer, a numeric variable, or a SAS expression and must be within the lower and upper bounds of the DIMENSION of the array

Program 6 is a modified version of Program 1 by using array processing.

Program 6:
```
data sbp2 (drop=i);
    set sbp;
    array sbparray [6] sbp1 sbp2 sbp3 sbp4 sbp5 sbp6;
    do i = 1 to 6;
        if sbparray [i] = 999 then sbparray [i] = .;
    end;
run;
```

THE DIM FUNCTION
You can use the DIM function to determine the number of elements in an array. The DIM function has the following form:

```
DIM[ARRAYNAME]
```

Using the DIM function is very convenient when you don't know the exact number of elements in your array, especially when you use _NUMERIC_, _CHARACTER_, and _ALL_ as array ELEMENTS. *Program 6* can be re-written by using the DIM function.

Program 7:
```
data sbp3 (drop=i);
    set sbp;
    array sbparray [*] sbp1 - sbp6;
    do i = 1 to dim(sbparray);
        if sbparray [i] = 999 then sbparray [i] = .;
    end;
run;
```

ASSIGNING INITIAL VALUES TO AN ARRAY
When creating a group of variables by using the ARRAY statement, you can assign initial values to the array elements. 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 DATA step variables. The $ is necessary since CHR1, CHR2, and CHR3 are not previously defined in the DATA step.

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

TEMPORARY ARRAYS

Temporary arrays contain temporary data elements. Using temporary arrays is useful when you want to create an array only for calculation purposes. When referring to a temporary data element, you refer to it by the ARRAYNAME and its DIMENSION. You cannot use the asterisk with temporary arrays. The temporary data elements are not output to the output dataset. The temporary data values are always automatically retained. To create a temporary array, you need to use the keyword _TEMPORARY_ as the array ELEMENT. For example, the following 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);
```

COMPILATION AND EXECUTION PHASES FOR ARRAY PROCESSING

COMPILATION PHASE

During the compilation phase, the PDV is created (_ERROR_ is omitted for simplicity purposes)

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:								
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

The array name SBPARRAY and array references are not included in the PDV. Each variable, SBP1 – SBP6, is referenced by the ARRAY reference. Syntax errors in the ARRAY statement will be detected during the compilation phase.

EXECUTION PHASE

Each step of the execution phase for *Program 6* is listed below:

- First iteration of the DATA step execution
  - At the beginning of the execution phase
    - _N_ is set to 1 in the PDV
    - The rest of the variables are set to *missing*

```
data sbp2 (drop=i);
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	1	•	•	•	•	•	•	•
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

  - The SET statement copies the first observation from *Sbp* to the PDV

```
set sbp;
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	1	141	142	137	117	116	124	•
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

  - The ARRAY statement is a compile-time only statement

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

  - First iteration of the DO loop
    - The INDEX variable I is set to 1

```
do i = 1 to 6;
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	1	141	142	137	117	116	124	1
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

- The array reference SBPARRAY[i] becomes SBPARRAY [1]
- SBPARRAY [1] refers to the first array element, SBP1
- Since SBP1 ≠ 999, there is no execution

```
if sbparray [i] = 999 then sbparray [i] = .;
```

- o SAS reaches the end of the DO loop
- o The rest of the iterations of the DO loop are processed the same as the first iteration
- o SAS reaches the end of the first iteration of the DATA step
  - The implicit OUTPUT writes the contents from the PDV to dataset *Sbp2*
  - SAS returns to the beginning of the DATA step

➢ Second iteration of the DATA step

- o At the beginning of the second iteration
  - _N_ is incremented to 2
  - SBP1 – SBP6 retained their values since these are the variables in both input and output datasets
  - I is set to *missing* since I is not in the input dataset

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	2	141	142	137	117	116	124	●
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

- o The SET statement copies the second observation from *Sbp* to the PDV

```
set sbp;
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	2	999	141	138	119	119	122	●
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

- o First iteration of the DO loop
  - The INDEX variable I is set to 1

```
do i = 1 to 6;
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	2	999	141	138	119	119	122	1
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

- The array reference SBPARRAY[i] becomes SBPARRAY [1]
- SBPARRAY [1] refers to the first array element, SBP1
- Since SBP1 = 999, SBP1 is set to *missing*

```
if sbparray [i] = 999 then sbparray [i] = .;
```

	_N_ (D)	SBP1 (K)	SBP2 (K)	SBP3 (K)	SBP4 (K)	SBP5 (K)	SBP6 (K)	I (D)
PDV:	2	●	141	138	119	119	122	1
		↑ SBPARRAY[1]	↑ SBPARRAY[2]	↑ SBPARRAY[3]	↑ SBPARRAY[4]	↑ SBPARRAY[5]	↑ SBPARRAY[6]	

- o SAS reaches the end of the DO loop
- o The rest of the iterations of the DO loop are processed the same as the first iteration
- o SAS reaches the end of the first iteration of the DATA step
  - The implicit OUTPUT writes the contents from the PDV to dataset *Sbp2*
  - SAS returns to the beginning of the DATA step

➢ The rest of the iterations of the DATA step are processed the same as above

APPLICATIONS BY USING THE ONE-DIMENSIONAL ARRAY

CREATING A GROUP OF VARIABLES BY USING ARRAYS

The *Sbp2* dataset contains 6 measurements of SBP for 4 patients. *Sbp2* has 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

Suppose that the first three measurements are the results based on the pre-treatment results and the last three measures are based on the post-treatment results. Suppose that the average SBP values for the pre-treatment measurements is 140 and the average SPB is 120 for the measurement after the treatments. You would like to create a list of variables ABOVE1 – ABOVE6, which indicates whether each measure is above (1) or below (0) the average measurement. The solution is in *Program 8.*

Program 8:
```
data sbp4 (drop=i);
    set sbp2;
    array sbp[6];
    array above[6];
    array threshhold[6] _temporary_ (140 140 140 120 120 120);
    do i = 1 to 6;
        if (not missing(sbp[i]))
        then above [i] = sbp[i] > threshhold[i];
    end;

proc print data=sbp4;
run;
```

Output:
```
Obs  sbp1  sbp2  sbp3  sbp4  sbp5  sbp6  above1  above2  above3  above4  above5  above6

 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
```

The first ARRAY statement in *Program 8* is used to group the existing variables, SBP1 – SBP6. The second ARRAY statement creates 6 new DATA step variables, ABOVE1 – ABOVE6. The third ARRAY statement creates temporary data elements that are used only for comparison purposes.

THE IN OPERATOR

Suppose that you would like to determine whether a specific value exits in a list of variables. For example, you would like to know whether SBP1 – SBP6 contains missing values. To achieve this task, you can utilize ARRAY processing with a DO loop.

Program 9:
```
data sbp5 (drop=i);
    set sbp2;
    array sbp [6];
    miss = 0;
    do i = 1 to dim(sbp);
        if missing(sbp[i]) then miss = 1;
    end;

proc print data=sbp5;
run;
```

Output:

```
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
```

Instead of using the DO loop, you can use the IN operator with ARRAYNAME. The IN operator can be used with either character or numeric arrays. For example,

Program 10:
```
data sbp6;
    set sbp2;
    array sbparray [*] sbp1 - sbp6;
    if . IN sbparray then miss = 1;
    else miss = 0;
run;
```

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, to calculate the product of NUM1 – NUM4 in the *test* data set, you can utilize array processing like *Program 11*.

Test:

	Num1	Num2	Num3	Num4
1	4	.	2	3
2	.	2	3	1

Program 11:
```
data product (drop=i);
    set test;
    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;
```

RESTRUCTURING DATASETS USING ARRAYS

Transforming a data set with one observation per subject to multiple observations per subject or vice versa is one of the common tasks for SAS programmers. Suppose that you have the following two data sets. The data set *wide* contains one observation for each subject and data set *long* contains multiple observations for each subject.

Wide:

	ID	S1	S2	S3
1	A01	3	4	5
2	A02	4	.	2

Long:

	ID	TIME	SCORE
1	A01	1	3
2	A01	2	4
3	A01	3	5
4	A02	1	4
5	A02	3	2

Program 12:
```
data long (drop=s1-s3);
    set wide;
    time = 1;
    score = s1;
    if not missing(score) then output;
    time = 2;
    score = s2;
    if not missing(score) then output;
    time = 3;
    score = s3;
    if not missing(score) then output;
run;
```

Program 12 transforms data set *wide* to *long* without using array processing. This program is not efficient, especially if you have large numbers of variables that need to be transformed. Grouping all the numeric variables together by using an array and creating the SCORE variable inside a DO loop will make this task more efficient. *Program 13* is a modified version of *Program 12* by using array processing.

Program 13:
```
data long_1 (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 14 transforms dataset *long* to *wide* without using array processing. You only need to generate one observation once all the observations for each subject have been processed (that is why we need the '**`if lastd.id`**' statement). The newly-created variables S1 – S3 in the final dataset need to retain their values; otherwise S1 – S3 will be initialized to missing at the beginning of each iteration of the DATA step processing. Subject A02 is missing one observation for TIME equaling 2. The value of S2 from the previous subject (A01) will be copied to the dataset *wide* for the subject A02 instead of a missing value because S2 is being retained. Thus, initialize S1 – S3 to *missing* when processing the first observation for each subject.

Program 14:
```
proc sort data=long;
    by id;

data wide (drop=time score);
    set long;
    by id;
    retain s1 - s3;
    if first.id then do;
        s1 = .;  s2 = .; s3 = .;
    end;
    if time = 1 then s1 = score;
    else if time = 2 then s2 = score;
    else s3 = score;
    if last.id;
run;
```

Program 15 is a modified version of *Program 14* by using array processing.

Program 15:
```
proc sort data=long;
    by id;

data wide_1 (drop=time score i);
    set long;
    by id;
    array s[3];
    retain s1 - s3;
    if first.id then do;
        do i = 1 to 3;
            s[i] = .;
        end;
    end;
    s[time] = score;
    if last.id;
run;
```

MULTIDIMENSIONAL ARRAYS

The syntax for creating multidimensional arrays is similar to the one for creating one-dimensional arrays. The only difference is using multiple numbers instead of one number for the array DIMENSION.

ARRAY ARRAYNAME[R, C, …] <$> <ELEMENTS>;

In the ARRAY statement, R refers to the number of rows and C to the number of columns. If there are three dimensions, the next number will refer to the number of pages. For example, the following array groups SBP1 – SBP6 into a two-dimensional array.

```
array sbp[2,3] sbp1 – sbp6;
```

	Columns		
Rows	SBP1	SBP2	SBP3
	SBP4	SBP5	SBP6

To reference an element in a two-dimensional array, you need to use both row and column indices. For example, to reference SBP3, you need to write SBP[1,3].

APPLICATIONS BY USING THE MULTIDIMENSIONAL ARRAY

CALCULATING AVERAGE SBP FOR PRE- AND POST- TREATMENT

Recall that the *Sbp2* dataset contains three pre-treatment measurements of SBP and three post-treatment measurements of SBP values. Suppose that you would like to create a dataset that contains average SPB for pre- and post- treatment for each patient. One way to solve this problem is to use a two-dimensional array as in *Program 16*.

Program 16:
```
data sbp7 (drop = i j sbp1 - sbp6);
    set sbp2;
    array sbp [2, 3];
    array sbpmean [2];
    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;
```

Program 16 uses a 2 by 3 two-dimensional array to group SBP1 – SBP6; the first row of the array will contain the pre-treatment results, S1 - S3 in the second row will contain the post-treatment results. There will be two variables that contain the mean measures for pre- and post- measures; 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 which are 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 measure and one for post-treatment measure. There are three iterations for the inner loop since there are three measurements for the pre- and post- measurements.

RESTRUCTURING DATASETS BY USING THE MULTIDIMENSIONAL ARRAY

The data set *Dat1* contains two records for each person, whereas the data set *Dat2* 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
3	2	B	A	D	C	B	C

Program 17 transforms data set *Dat1* to data set *Dat2*. In this program, since you are only creating the observation after you finish reading all the observations for each person, you need to use the BY-group processing by using ID as the BY-variable. The output will be 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 that corresponds to the number of observations for each subject. The second dimension for ALL_G[2, 3] is 3 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 the index *i* within each iteration of the DATA step. The iterated DO loop is used to increment the index *j* within each iteration of the DATA step execution. The newly-created variables M_G1, M_G2, M_G3, F_G1, F_G2, and F_G3 need to be retained via the RETAIN statement by preventing them to be set to missing at the beginning of each iteration.

Program 17:
```
proc sort data=dat1;
    by id;

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 m_g1 - m_g3 f_g1 - f_g3;
    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 18 transforms data set *Dat2* to data set *Dat1*. The idea behind *Program 18* is similar to *Program 17.* 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 within the outer loop but outside the inner loop is needed to create two observations for each iteration of the outer loop.

Program 18:
```
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;
    end;
run;
```

CONCLUSION

Since array processing enables you to perform the same tasks for a group of related variables, it allows you to create more efficient programming code. However, in order to use arrays correctly, in addition to grasping the array syntax and its complex rules, you also need to understand how DATA steps are processed. In the end, you will often realize that most of the errors are closely related to programming fundamentals, which is understanding how the PDV works.

REFERENCES

SAS Institute Inc. 2006. SAS OnlineDoc® 9.1.3. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Arthur Li
City of Hope Comprehensive Cancer Center
Division of Information Science
1500 East Duarte Road
Duarte, CA 91010 - 3000
Work Phone: (626) 256-4673 ext. 65121
Fax: (626) 471-7106
E-mail: xueli@coh.org