PharmaSUG 2013 – Paper HT04

Effectively Utilizing Loops and Arrays in the DATA Step

Arthur X. Li, City of Hope National Medical Center, Duarte, CA

ABSTRACT

The implicit loop refers to the DATA step repetitively reading data and creating observations, one at a time. The explicit loop, which utilizes the iterative DO, DO WHILE, or DO UNTIL statements, is used to repetitively execute certain SAS® statements within each iteration of the DATA step execution. Utilizing explicit loops is often used to simulate data and to perform a certain computation repetitively. However, when an explicit loop is used along with array processing, the applications are extended widely, which includes transposing data, performing computations across variables, etc. Being able to write a successful program that uses loops and arrays, one needs to know the contents in the program data vector (PDV) during the DATA step execution, which is the fundamental concept of DATA step programming. This workshop will cover the basic concepts of the PDV, which is often ignored by novice programmers, and then will illustrate how utilizing loops and arrays to transform lengthy code into more efficient programs.

INTRODUCTION

A loop is one of the basic logic programming language structures that allows us to execute one or a group of statements repetitively until it reaches a predefined condition. This type of programming language construction is widely used in all computer languages. Compared to other programming languages, understanding the necessities of creating loops is more complex for the SAS language since there are implicit and explicit loops and sometimes beginning programmers can't distinguish clearly between them. Knowing when the time is right to create an explicit loop is one of the challenges that face all beginning programmers. Furthermore, unlike other programming language, an array in SAS is not a type of data structure; instead, an array is used to temporarily group a set of variables. Grouping variables by using an array allows you to modify data more efficiently mostly because you will be able to utilize loops to work along the grouped variables.

IMPLICIT AND EXPLICIT LOOPS

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 Program Data Vector (PDV) is created according to the descriptor portion of the input dataset.

The execution phase starts after the compilation phase. In the execution phase, SAS uses the PDV to build the new data set. Not all of the variables in the PDV are outputted to the final data set. The variables in the PDV that are flagged with "K" (which stands for "kept") will be written to the output dataset; on the other hand, the variables flagged with "D" (which stands for "dropped") will not.

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 in a clinical trial where each patient has a 50% chance of receiving either the drug or a placebo. 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 1*.

Patient.sas7bdat

	ID
1	M2390
2	F2390
3	F2340
4	M1240

Program 1:

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

Output:

Obs	ID	group
1	M2390	P
2	F2390	D
3	F2340	D
4	M1240	D

FIRST ITERATION:

Patient.sas7bdat:

	ID
1	M2390
2	F2390
3	F2340
4	M1240

← Reading

End-of-file marker →

```
data example1 (drop=rannum);
```

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

EXPLANATION: _N_ is initialized to 1 and _ERROR_ is initialized to 0. The variables ID, GROUP, and RANNUM are set to missing.

```
set patient;
```

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

EXPLANATION: The SET statement copies the first observation from *patient* to the PDV.

```
rannum = ranuni(2);
```

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

EXPLANATION: RANNUM is generated by the RANUNI function.

```
if  rannum> 0.5 then group = 'D';  else group ='P';
```

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

EXPLANATION: Since RANNUM is not greater than 0.5, GROUP is assigned with value 'P.'

```
run;
```

Example1:

ID	GROUP
M2390	P

EXPLANATION: The implicit OUTPUT statement at the end of the DATA step tells SAS to write observations to the dataset. Since SAS didn't read the end-of-file marker, it returns to the beginning of the DATA step to begin the 2nd iteration.

Figure 1. The first iteration of Program 1.

At the beginning of the execution phase, the automatic variable _N_ is initialized to 1 and _ERROR_ is initialized to 0. _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 (See Figure 1). Next, the SET statement copies the first observation from the dataset *patient* to the PDV. Then the variable RANNUM is

generated by the **RANUNI**[1] function. Since RANNUM is not greater than 0.5, GROUP is assigned with value 'P'. The implicit OUTPUT statement at the end of the DATA step tells SAS to write the contents from the PDV that is marked with "K" 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, 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[2]. 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 data set 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 2:

```
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;
```

The DATA step in *Program 2* begins with assigning ID numbers and then assigns the group value based on

---

[1] 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.

[2] 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.

a generated random number. There are four explicit OUTPUT statements[3] that tells 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 2

Program 3:

```
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;
run;
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 number, say 1 to 4, as patient IDs; you can rewrite the previous program as below:

Program 4:

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

Output:

Obs	id	group
1	1	P
2	2	D
3	3	D
4	4	D

Program 4 didn't specify the INCREMENT value, thus the default value 1 is used. You can also decrement a DO loop by using a negative value, such as -1. The execution phase is illustrated in figure 2a and 2b.

[3] 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

```
data example4 (drop=rannum);
```

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

EXPLANATION: _N_ is initialized to 1 and _ERROR_ is initialized to 0. The variables ID, GROUP, and RANNUM are set to missing.

FIRST ITERATION OF THE DO LOOP:

```
do id = 1 to 4;
```

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

EXPLANATION: ID is assigned to 1 at the beginning of the first DO loop.

```
rannum = ranuni(2);
```

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

EXPLANATION: RANNUM is generated by the RANUNI function.

```
If rannum> 0.5 then group = 'D';  else group ='P';
```

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

EXPLANATION: Since RANNUM is not greater than 0.5, GROUP is assigned with value 'P.'

```
output;
```

Example3:

ID	GROUP
1	P

EXPLANATION: The OUTPUT statement tells SAS to write observations to *Example3*. SAS reaches the end of the DO loop.

SECOND ITERATION OF THE DO LOOP:

```
do id = 1 to 4;
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	2	0.3699251396	P

EXPLANATION: ID is incremented to 2; SINCE 2 IS ≤ 4, the 2nd iteration continues.

```
rannum = ranuni(2);
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	2	0.9401774313	P

EXPLANATION: RANNUM is generated by the RANUNI function.

```
if rannum> 0.5 then group = 'D';  else group ='P';
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	2	0.9401774313	D

EXPLANATION: Since RANNUM is greater than 0.5, GROUP is assigned with value 'D.'

```
output;
```

Example3:

ID	GROUP
1	P
2	D

EXPLANATION: The OUTPUT statement tells SAS to write observations to *Example3*. SAS reaches the end of the DO loop.

```
run;
```

Figure 2a. The first two iterations of the DO loop in Program 4.

THIRD ITERATION OF THE DO LOOP:

```
do id = 1 to 4;
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	3	0.9401774313	D

EXPLANATION: ID is incremented to 3; since 3 is ≤ 4, the 3rd iteration continues.

```
rannum = ranuni(2);
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	3	0.7996486122	D

EXPLANATION: RANNUM is generated by the RANUNI function.

```
if rannum> 0.5 then group = 'D';  else group ='P';
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	3	0.7996486122	D

EXPLANATION: Since RANNUM is greater than 0.5, GROUP is assigned with value 'D.'

```
output;
```

Example3:

ID	GROUP
1	P
2	D
3	D

EXPLANATION: The OUTPUT statement tells SAS to write observations to *Example3*. SAS reaches the end of DO loop.

FOURTH ITERATION OF THE DO LOOP:

```
do id = 1 to 4;
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	4	0.7996486122	D

EXPLANATION: ID is incremented to 4; since 4 is ≤ 4, the 4th iteration continues.

```
rannum = ranuni(2);
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	4	0.5187972908	D

EXPLANATION: RANNUM is generated by the RANUNI function.

```
if rannum> 0.5 then group = 'D';  else group ='P';
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	4	0.5187972908	D

EXPLANATION: Since RANNUM is greater than 0.5, GROUP is assigned with value 'D.'

```
output;
```

Example3:

ID	GROUP
1	P
2	D
3	D
4	D

EXPLANATION: The OUTPUT statement tells SAS to write observations to *Example3*. SAS reaches the end of DO loop.

```
end;
```

	_N_ (D)	_ERROR_ (D)	ID (K)	RANNUM (D)	GROUP(K)
PDV:	1	0	5	0.5187972908	D

EXPLANATION: ID is incremented to 5; since 5 is > 4, the loop ends.

```
run;
```

EXPLANATION: There will be no explicit OUTPUT statement. Since we didn't read an input dataset, the DATA step execution ends.

Figure 2b. The last two iterations of the DO loop in Program 4.

EXECUTING DO LOOPS CONDITIONALLY

Program 4 used an iterative DO loop, which requires that you specify the *number* of iterations for the DO loop. Sometimes you will need to execute statements repetitively until a condition is met. In this situation, you need to use either the DO WHILE or DO UNTIL statements. Following is the syntax for the DO WHILE statement:

```
DO WHILE (EXPRESSION);
SAS STATEMENTS
END;
```

In the DO WHILE loop, the EXPRESSION is evaluated at the top of the DO loop. The DO loop will not execute if the EXPRESSION is false. We can rewrite *program 4* by using the DO WHILE loop.

Program 5:

```
data example5 (drop=rannum);
    do while (id <4);
        id + 1;
        rannum = ranuni(2);
        if  rannum> 0.5 then group = 'D';
        else group ='P';
        output;
    end;
run;
```

In *program 5*, the ID variable is created inside the loop by using the SUM statement[4]. Thus, the variable ID is initialized to 0 in the PDV at the beginning of the DATA step execution, which is before the DO WHILE statement. At the beginning of the loop, the condition is being checked. Since ID equals 0, which satisfies the condition (ID < 4), the first iteration begins. For each iteration, the ID variable is accumulated from the SUM statement. The iteration will be processed until the condition is not met.

Alternatively, you can also use the DO UNTIL loop to execute the statements conditionally. Unlike DO WHILE loops, the DO UNTIL loop evaluates the condition at the end of the loop. That means the DO UNTIL loop always executes at least once. The DO UNTIL loop follows the form below:

```
DO UNTIL (EXPRESSION);
SAS STATEMENTS
END;
```

The following program is based on *program 4* by using the DO UNTIL loop.

Program 6:

```
data example6 (drop=rannum);
    do until (id >=4);
        id +1;
        rannum = ranuni(2);
        if rannum > 0.5 then group = 'D';
        else group ='P';
        output;
    end;
run;
```

NESTED LOOPS

You can place a loop within another loop. To continue with the previous example, suppose that you would like to assign 12 subjects from 3 cancer centers (“COH”, “UCLA”, and “USC”) with 4 subjects per center, where each patient has a 50% chance of receiving either the drug or a placebo. A nested loop can be used to solve this problem. In the outer loop, the INDEX-VARIABLE, CENTER, is assigned to the values with the

[4] The variable that is created by the SUM statement is automatically set to 0 at the beginning of the first iteration of the DATA step execution and it is retained in following iterations.

name of the three cancer centers. For each iteration of the outer loop, there is an inner loop that is used to assign each patient to a group.

Program 7:

```
data example7;
   length center $4;
   do center = "COH", "UCLA", "USC";
      do id = 1 to 4;
         if ranuni(2) > 0.5 then group = 'D';
         else group ='P';
         output;
      end;
   end;
run;
proc print data=example7;
run;
```

Output:

Obs	center	id	group
1	COH	1	P
2	COH	2	D
3	COH	3	D
4	COH	4	D
5	UCLA	1	D
6	UCLA	2	D
7	UCLA	3	P
8	UCLA	4	P
9	USC	1	P
10	USC	2	P
11	USC	3	D

COMBINING IMPLICIT AND EXPLICIT LOOPS

In *example 7*, all the observations were created from one DATA step since we didn't read any input data. Sometimes it is necessary to use an explicit loop to create multiple observations for each observation that is read from an input dataset. In *example 7*, we used an outer loop to create a CENTER variable. Suppose the values for CENTER is stored in a SAS dataset. For each center, you need to assign 4 patients where each patient has a 50% chance of receiving either the drug or a placebo. Following is the program to solve this problem:

Cancer_center.sas7bdat

	CENTER
1	COH
2	UCLA
3	USC

Program 8:

```
data example8;
    set cancer_center;
    do id = 1 to 4;
        if  ranuni(2)> 0.5 then group = 'D';
        else group ='P';
        output;
    end;
run;
```

Program 8 is an example of using both implicit and explicit loops. The DATA step is an implicit loop that is used to read observations from the input dataset, *Cancer_center.* For each CENTER that is being read, there is an explicit loop to assign patients to either 'D' or 'P.'

APPLICATIONS OF USING LOOPS

SIMULATION

To generate a random number that follows standard normal distribution (mean = 0, standard deviation = 1), you can use the RANNOR function. To generate a random number that follows a normal distribution with mean equaling *mu* and standard deviation equaling *sd*, you can use this formula: mu + sd*rannor(seed).

Program 9 utilizes an iterative DO loop to simulate data for 1000 subjects. The ID variable serves as the index-variable in the loop. Within each iteration of the DO loop, if the number that is generated form RANUNI is less than 0.5, the simulated subject will be assigned to group 'A' with the simulated weight following the normal distribution with mean = 200 and standard deviation equals 15; otherwise, subjects that are assigned to group 'B,' their mean weight will be 250 and standard deviation will equal 50. The MEANS procedure is used to validate whether the data are simulated correctly.

Program 9:

```
data example9;
    do id = 1 to 1000;
        if ranuni(2) < 0.5 then do;
            group = 'A';
           weight = 200 + 15*rannor(2);
        end;
        else do;
            group = 'B';
            weight = 250 + 50*rannor(2);
        end;
        output;
    end;
run;

proc means data= example9 n mean std;
    class group;
    var weight;
run;
```

Output:

```
The MEANS Procedure

             Analysis Variable : weight

             N
group     Obs       N            Mean         Std Dev
_____________________________________________________
A         490     490     200.0780187      15.8693403

B         510     510     245.8652661      51.9449135
_____________________________________________________
```

CALCULATING BALANCE FOR INVESTMENT

Suppose that you would like to invest $1,000 each year at one bank. The investment earns 5% annual interest, compounded monthly (that means the interest for each month will be 0.05/12).

Program 10 uses nested iterative DO loops to calculate the balance for each month if you are investing for 2 years. In the outer loop, the YEAR variable serves as the index-variable. Within the outer loop, the CAPITAL variable is accumulated by using the SUM statement and an iterative DO loop (the inner loop) is used to calculate the interest and the capital for each month.

Program 10:

```
Data example10;
    do year = 1 to 2;
        capital + 1000;
        do month = 1 to 12;
            interest = capital*(0.05/12);
            capital+interest;
            output;
        end;
    end;
run;
```

INTRODUCTION TO ARRAY PROCESSING

THE PURPOSE OF USING ARRAYS

The data set SBP 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 11*.

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 11:

```
data example11;
    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;
```

Notice that each of the IF statements in *Program 11* 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.

ARRAY DEFINITION AND SYNTAX FOR ONE DIMENSIONAL ARRAY

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 12 is a modified version of Program 11 by using array processing.

Program 12a:

```
data example12a (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 12b:

```
data example12b (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

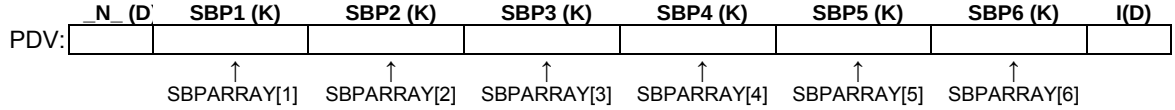
Temporary arrays contain temporary data elements. Using temporary arrays is useful when you want to create an array only for calculation purposes. When referring to a temporary data element, you refer to it by the ARRAYNAME and its DIMENSION. You cannot use the asterisk with temporary arrays. The temporary 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)



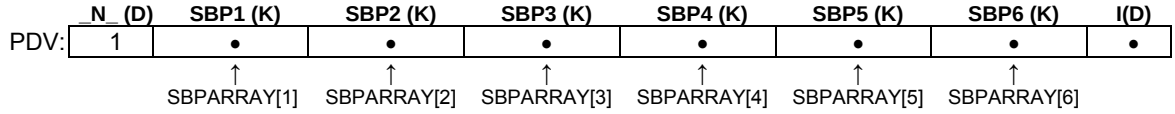
The array name SBPARRAY and array references are not included in the PDV. Each variable, SBP1 – SBP6, is referenced by the ARRAY reference. Syntax errors in the ARRAY statement will be detected during the compilation phase.

EXECUTION PHASE

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

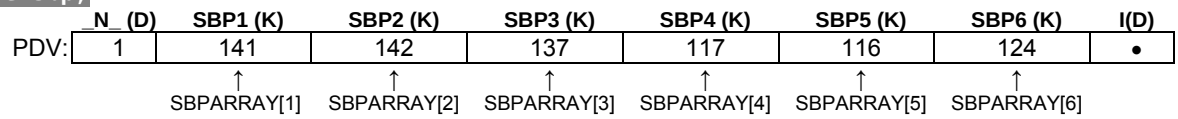
- First iteration of the DATA step execution
  - At the beginning of the execution phase
    - _N_ is set to 1 in the PDV
    - The rest of the variables are set to *missing*

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



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

```
set sbp;
```



  - The ARRAY statement is a compile-time only statement

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

<Effectively Utilizing Loops and Arrays in the DATA Step>, continued

- 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] = .;
```

- SAS reaches the end of the DO loop
- The rest of the iterations of the DO loop are processed the same as the first iteration
- 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
- 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]	

- 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]	

- 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]	

- SAS reaches the end of the DO loop
- The rest of the iterations of the DO loop are processed the same as the first iteration
- 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 13:

```
data example13 (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;
run;

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. You can use the IN operator with ARRAYNAME. The IN operator can be used with either character or numeric arrays. For example,

Program 14:

```
data example14 (drop=i);
    set sbp2;
    array sbp[6];
    if . IN sbp then miss = 1;
    else miss = 0;
run;

proc print data=example14;
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

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*.

Product:

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

Program 15:

```
data example15 (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 16:

```
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 16 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 17* is a modified version of *Program 16* by using array processing.

Program 17:

```
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 18 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 18:

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

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 19 is a modified version of *Program 18* by using array processing.

Program 19:

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

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;
```

CONCLUSION

Loops are important programming language structures that allow you to create more simplified and efficient programming codes. 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 as well. However, in order to use loop structures and arrays correctly, in addition to grasping the 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

Li, Arthur. 2013. Handbook of SAS® DATA Step Programming. Chapman and Hall/CRC.

CONTACT INFORMATION

Arthur Li
City of Hope National Medical 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: arthurli@coh.org