ARRAYS: KNOW WHEN TO HOLD THEM, KNOW WHEN TO FOLD THEM.

Patricia Hettinger, Certified SAS® Professional, Oakbrook Terrace, IL

ABSTRACT

Sometimes you have to process data as a group, with one item of the group influencing how the other items are processed (think draw poker or bowling scores). This is often best handled by using SAS variable arrays. This paper goes through a complete application from loading the arrays to using one to influence other variables and tips on testing and validation.

INTRODUCTION

An array can be defined as a data structure in which similar elements of data are arranged in a table. Many other programming languages such as COBOL let you treat an array either as the entire structure or as individual elements. This is not the case with SAS which generally will only let you work with individual elements. There is an exception to this which we will see in the testing section. The array is only available in the data step and cannot be referred to in any procedure.

DEFINING ARRAYS

The ARRAY statement has two required parameters, array name and a subscript in the format ARRAY array_name (subscript) in which array_name is the name the group can be identified by and the subscript is the number of elements. Although the array_name itself will not be retained past the data step, all the numbered elements will be in the output. For example, say we create an array named tests with five elements as ARRAY tests(5). The output will have five variables, tests1 tests2, tests3, tests4 and tests5. If we do nothing else in the step, all five variables will have missing values. The only way to load a SAS array is load it element by element. In other words, the statement tests=12345 will give you an error while tests(1) = 1 will populate tests1.

Of course, we don't have to settle for the name SAS gives our variables. We can name our individual elements anything we want as long as they follow SAS naming conventions. For example, we could name the same array like this:

```
array tests(5) quarter1 – quarter5;
```

The resulting output would have quarter1 through quarter5 instead of tests1 through tests5. We could even name them something completely different:

```
array tests(5) biology botany anatomy chemistry physics;
```

and get the five STEM variables in our output. If you go this route, you must name the same number of variables as your subscript. You will get an error message if you err in either direction:

```
array tests(5) quarter1-quarter4;
```

```
ERROR: Too few variables defined for the dimension(s) specified for the array tests.
```

```
array tests(5) quarter1-quarter6;
```

```
ERROR: Too many variables defined for the dimension(s) specified for the array tests.
```

If you want SAS to keep track of your variables, you may use the format ARRAY array_name (*) or ARRAY array_name. For our tests array, either would be equivalent:

```
array tests(*) quarter1-quarter10   -OR-
array tests quarter1-quarter10
```

To get the number of elements for future use, the function DIM will give the correct count, dim(tests) = 10.

Another way you can define your array to start your subscript as somewhere other than one, the default. You may never want to do this but you should know it is possible.

This can be done in our tests case with:

```
array tests(2:5) ;
```

You will see variables tests1, tests2, tests3 and tests4 in the output. There will be no tests5. If you want to refer to test1, you would use '2' as the value for subscript, '3' for test2 and so on.

The above examples were all numeric. What if we want a character array? To have a character array instead of the default numeric array, we need two more parameters in the ARRAY statement, '$' and the length, ARRAY array_name(subscript) $ length. We would code this to turn our tests array into one containing five elements with character length 10:

```
array tests(5) $10. quarter1-quater5;
```

SAS VARIABLES _ALL_, _NUMERIC_, _CHARACTER_ and _TEMPORARY_

Like elsewhere in the SAS language, you can use the _ALL_ statement in an array definition. The _ALL_ variable in this context, puts ALL of the incoming data set variables into the array, assuming that they are all of the same type. The _NUMERIC_ variable will put all numeric variables into the array while _CHARACTER_ will put all character variables. Consider the mixed type data set 'property' in illustration 1:

property_id	city	rate_sq_foot
45	Chicago	40.00
46	Dallas	30.00
47	Kansas City	25.00

Illustration 1

We can attempt to put all the variables into the array rates with code like this:

```
data rates;
set property;
array sq_fee_rates(*) _ALL_;
```

This will not work at all with the message:

```
ERROR: All variables in array list must be the same type, i.e., all numeric or character.
```

If we used the _NUMERIC_ variable, this error message will not occur:

```
data rates;
set property;
array sq_fee_rates(*) _NUMERIC_;
```

There will be two elements in the sq_fee_rates array, property_id and rate_sq_foot. You may refer to the elements by either those names or by sq_fee_rates (subscript 1) and sq_fee_rates(subscript 2).

Subscript 1 Subscript 2

property_id	rate_sq_foot
45	40.00
46	30.00
47	25.00

Illustration 2

The _CHARACTER_ variable works in a similar fashion with just the city variable going into the array:

```
data rates;
set property;
array sq_fee_rates(*) _CHARACTER_;
```

Subscript 1

city
Chicago
Dallas
Kansas City

Illustration 3

If you ever use these variables, define it with the asterisk as the subscript value to avoid getting the too few or too many variable definition error.

One variable you probably will use is the _TEMPORARY_ variable. This ensures that your array variables do not survive the data step in they have been created. Examine some code creating a temporary array named tests:

```
array tests(5) _TEMPORARY_;
```

Notice that there were no named elements. SAS does not allow this for a temporary array. Nor does SAS allow the asterisk in the subscript. A temporary array must have the number of elements described at the very beginning.

MULTIDIMENSIONAL ARRAYS

We can describe arrays with more than one dimension as well. We can think of these as having rows and columns. There are three major differences in describing a multiple as opposed to single dimensional array. One is you use two or more subscripts in the format ARRAY array_name(subscript1,subscript2,etc.). Another difference is that the number of variables output is equal to the subscripts multiplied by each other. The last major difference is you cannot use the asterisk for either subscript. They must be integer numbers. As with a single dimension array, all the elements must be numeric or all character. There can be no mixture.

Defining an array called multitable as -

```
array multitable(2,10,10);
```

would result in 200 elements ranging from multitable1 to multitable200 (2 X 10 X 10). If we wanted to name these variables as in the tests array before, we would have to come up with 200 names.

POPULATING YOUR ARRAY

Now that we have our array defined, what do we do with it? Simply declaring an array will not populate the elements inside nor update them, if you are using a list of element names that exists already. And as stated before, SAS will not let you do a group move. You could update each element by name like quarter1=1, quarter2=2, etc. but this is tedious and defeats the purpose of an array. You will most likely use one of SAS DO loops constructs.

SINGLE DIMENSION ARRAY EXAMPLE

Let's say we want to take the list of people at a weekly weigh-in from having the records in a separate row for each participant to having all the records being in one row across multiple columns. This desire should be nothing new to those of us who work with spreadsheets. Consider the list in illustration 4:

participant_Id	week_ending	weight
1234	06/3/2017	200.50
1234	06/10/2017	195.00
1234	06/17/2017	194.50
1234	06/24/2017	193.00
1234	07/1/2017	192.00
5678	06/3/2017	155.00
5678	06/10/2017	152.50
5678	06/17/2017	152.00
5678	06/24/2017	150.50
5678	07/1/2017	148.00

Illustration 4

We would like the weights in five columns for each week in the program. We'd also like to calculate the amount lost each week. Let's see how far PROC TRANSPOSE can take us:

```
proc transpose data=weekly_record out=trans_weekly;
id week_ending; /*variable we want as the identifying variable*/
idlabel week_ending; /*use week_ending variable values for the labels*/
by participant_id; /*one row per participant*/
run;
```

Close but not there yet:

participant_id	_NAME_	_LABEL_	06-03-2017	06-10-2017	06-17-2017	06-24-2017	07-01-2017
1234	weight	weight	200.50	195.00	194.50	193.00	192.00
5678	weight	weight	155.00	152.50	152.00	150.50	148.00

Illustration 5

But hey, we have a set of data in numeric format in five columns labeled with the end of the appropriate week. We can do array processing. So let's define a few arrays and process them (with a few comments):

```
data weekly_loss;
set trans_weekly;
array weights(5) "06-03-2017"N '06-10-2017'N '06-17-2017'N '06-24-2017'n '07-01-2017'n; /*already exist*/
array weight_loss(4) '06-10-2017 Loss 'N '06-17-2017 Loss'N '06-24-2017 Loss'n '07-01-2017 Loss'n; /*new*/

/*remember to put variable names with embedded spaces or special characters surrounded by either single or double
quotes with a 'N'  or n suffix*/

do i = 1 to 4;
 /*why 4 and not 5?  We are calculating the weight loss from week to week.  Using 5 would give
a subscript out of range error*/
weight_loss(i) = weights(i) - weights(i+1); /*subtract current weight from previous weight*/
end;
put weights(*); /*only place where this reference to the entire arrays will work*/
put weight_loss(*);
run;
```

Some explanation should be given for the put weights(*) and put weight_loss(*) statements. This is about the only way a reference to the entire array will give results. If you try dropping or keeping your array variables, by referring to the entire array, you will receive an error message. For that matter, if you try dropping or keeping an array element by its subscript, you will get an error too.

```
drop weight_loss(*);
```

```
ERROR:  Syntax error, expecting on the following:  a name, -, :, ,, _ALL_,
_CHARACTER_, _CHAR_, _NUMERIC_
```

Even if you try the _ALL_ variable, you will not be able to do this. In this case, you will receive a much more informative error.

```
drop weight_loss _all_;
```

```
ERROR: The array weight_loss is not allowed in a DROP/KEEP/RENAME context.
```

Referring to the array in a put statement is allowed, however and is especially helpful for debugging your program. The log shows the output. The first and third lines have the weights recorded. The second and fourth have the calculated weight loss.

```
Line 1       200.5  195    194.5  193    192
Line 2              5.5    0.5    1.5    1
Line 3       155    152.5  152    150.5  148
Line 4              2.5    0.2    1.5    2.5
```

Based on the log, the program seems to be working correctly. We can either comment out the put statements or remove them completely for the next run.

Our output will now have variables '06-03-2017', 06-10-2017', '06-17-2017', '06-24-2017', '07-01-2017', 06-10-2017 Loss ', '06-17-2017 Loss', '06-24-2017 Loss' and '07-01-2017 Loss'.

participant_id	_NAME_	_LABEL_	06-03-2017	06-10-2017	06-17-2017	06-24-2017	07-01-2017
1234	weight	weight	200.50	195.00	194.50	193.00	192.00
5678	weight	weight	155.00	152.50	152.00	150.50	148.00

Plus

06-10-2017 Loss	06-17-2017 Loss	06-24-2017 Loss	07-01-2017 Loss	i
5.50	0.50	1.50	1.00	5
2.50	0.50	1.50	2.50	5

Illustration 6

SCRAMBLE – USING A MULTIDIMENSIONAL ARRAY

Have you ever had to scramble a name or number in your work? Perhaps you are sending account information to an outside vendor. Or you want an extra layer of security for data you are storing in the cloud. Using a multidimensional array is one way to handle this. We'll work out a basic schema in Excel first for a simple scramble program for a five position zip code. We will refer to the original zip code five elements as the start elements and the scrambled zip code elements as the target. We will move each start element over one position to the target element, beginning the move according to two parameters, PARM 1 and PARM 2. PARM 1 gives the location of the start element to be moved first and PARM 2 gives the target element where the start element will be moved. After either the start element or target element's position is five, we'll start over at one. For example, if PARM 1 is 5 and PARM 2 is 2, start element 5 will move to the target element 2 first. Since we have now moved the fifth start element, we go back to the first start element and move it to the next target element, 3. We continue with moving start element 2 to target element 4, start element 3 to the target element 5. Target element 5 has been populated so we'll start over with target element 1. Thus we move start element 4 to target element 1 and populate our final output.

Illustration 7 should make this a little clearer – the thickest line indicates the first start and target locations for 5 and 2 as the parameters. The value coming in is '01560" (far left) and the scrambled value going out (far right) is '60015'.

SCRAMBLE PARM 1 = 5 PARM 2 = 2

IN VALUE	START POSITION	1	2	3	4	5	
01560	ORIGINAL VALUE	0	1	5	6	0	
	MOVEMENT FROM START TO TARGET	ONE TO THREE	TWO TO FOUR	THREE TO FIVE	FOUR TO ONE	FIVE TO TWO	
	TARGET POSITION	1	2	3	4	5	OUT VALUE
	SCRAMBLED VALUE	6	0	0	1	5	***60015***

Illustration 7

We want to make sure this value can be unscrambled when needed. Let's now call PARM 1 and PARM 2 the keys. If we know the key is 52, we should be able to unscramble the number by doing the movements in reverse, start element 2 now goes to target 5, 3 to 1, 4 to 2, 5 to 3 and 1 to 4

UNSCRAMBLE PARM 1 = 5 PARM 2 = 2

IN VALUE	START POSITION	1	2	3	4	5	
60015	SCRAMBLED VALUE	6	0	0	1	5	
	MOVEMENT FROM IN TO OUT	ONE TO FOUR	TWO TO FIVE	THREE TO ONE	FOUR TO TWO	FIVE TO THREE	OUT VALUE
	TARGET POSITION	1	2	3	4	5	**01560**
	UNSCRAMBLED VALUE	0	1	5	6	0	

Illustration 8

Just to make sure our logic is sound, let's try another key code, 23. That means, we'll start by moving start element 2 to target element 3. Our input value is still '01560' but now our output value is '00156'. Note that we started over at start element 1 once start element 5 was moved to target 1:

SCRAMBLE PARM 1 = 2 PARM 2 = 3

IN VALUE	START POSITION	1	2	3	4	5	
01560	ORIGINAL VALUE	0	1	5	6	0	
	MOVEMENT FROM IN TO OUT	ONE TO TWO	TWO TO THREE	THREE TO FOUR	FOUR TO FIVE	FIVE TO ONE	
	TARGET POSITION	1	2	3	4	5	OUT VALUE
	SCRAMBLED VALUE	0	0	1	5	6	***00156***

Illustration 9

And using the 23 key, we can unscramble when we need to.

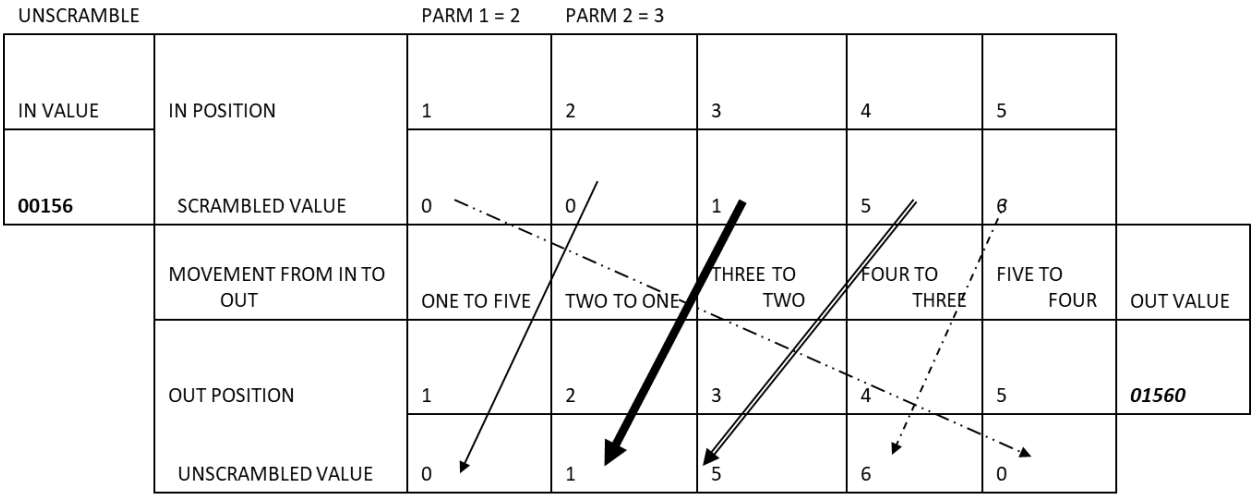

Illustration 10

We will code this using a two-dimensional array and test with output to the log until we get the loop working correctly. The loop coding and checking is probably the most difficult part of working with an array. Let's take a look at how we could use set this up to test the two-dimensional array storing the individual numbers of the zip code as well as the scrambled version:

```
data testscramble;
set ziplist;
array encrypt_zip(2,5) $1. current_zip1-current_zip5
            new_zip1 - new_zip5;
attrib zip_out format=$5.; /*output scrambled zip*/
SWAP1 = &PARM1; /*several ways to set PARM1 and PARM2 – SAS Enterprise Guide prompts were used here*/
SWAP2=&PARM2;
select (SWAP1);
 /*important setup for the second loop.  Note that for SWAP value '1', STOPLOOP2 is 0, meaning the second loop will not
execute */
            when (1) STOPLOOP2=0;
            when (2) STOPLOOP2=1;
            when (3) STOPLOOP2=2;
            when (4) STOPLOOP2=3;
            when (5) STOPLOOP2=4;
end;
/*initialization */
do i = 1 to 2;
            do j = 1 to 5;
if i = 1 then
            encrypt_zip(i,j) = substr(zip_in,j,1);
            else encrypt_zip(i,j) = 'X';
            end;
end;
put 'array at start ' encrypt_zip(*);
I=SWAP1;
J=SWAP2;
/*first loop will execute until we have finished processing start element 5*/
/*if parm1 equal to 1, this is the only loop that will execute*/
do until (I gt 5 );
            encrypt_zip(2,j) =encrypt_zip(1,i) ;
            put 'first loop: start element in position ' i ' is now ' encrypt_zip(1,i);
            put 'first loop: target element in position ' j ' is now '  encrypt_zip(2,j);
```

```
            put 'first loop - array after ' i ' and ' j ' processed ' encrypt_zip(*);
            substr(zip_out,j,1)=encrypt_zip(2,j);
            put 'first loop: zip_out is ' zip_out;
            I+1;
            J+1;
            if J gt 5 then J = 1;/*need this so target array doesn't go out of bounds*/
        end;
        /*if parm1 is greater than 1, this loop will execute too - it will stop at*/
        /*the predetermined stoploop2*/
        if SWAP1 gt 1 then do;
        if I gt 5 then I = 1;
        if J gt 5 then J = 1;
            do until (I gt STOPLOOP2);
                encrypt_zip(2,j) =encrypt_zip(1,I ) ;
                put 'stop second loop at ' stoploop2;
                put 'second loop: start element in position ' i ' is now ' encrypt_zip(1,i);
                put 'second loop: target element in position ' j ' is now '  encrypt_zip(2,j);
                put 'second loop - array after ' i ' and ' j ' processed ' encrypt_zip(*);
                substr(zip_out,j,1)=encrypt_zip(2,j);
                put 'second loop: zip_out is ' zip_out;
                I+1;
                J+1;
                if J gt 5 then J = 1;
            end;
        end;
        run;
```

When we run this code with our test data and key 23, we get this in the log for 01560:

```
array at start 0 1 5 6 0 X X X X X
first loop: start element in position 2  is now 1
first loop: target element in position 3  is now 1
first loop - array after 2  and 3  processed 0 1 5 6 0 X X 1 X X
first loop: zip_out is 1
first loop: start element in position 3  is now 5
first loop: target element in position 4  is now 5
first loop - array after 3  and 4  processed 0 1 5 6 0 X X 1 5 X
first loop: zip_out is 15
first loop: start element in position 4  is now 6
first loop: target element in position 5  is now 6
first loop - array after 4  and 5  processed 0 1 5 6 0 X X 1 5 6
first loop: zip_out is 156
first loop: start element in position 5  is now 0
first loop: target element in position 1  is now 0
first loop - array after 5  and 1  processed 0 1 5 6 0 0 X 1 5 6
first loop: zip_out is 0 156
stop second loop at 1
second loop: start element in position 1  is now 0
second loop: target element in position 2  is now 0
second loop - array after 1  and 2  processed 0 1 5 6 0 0 0 1 5 6
second loop: zip out is 00156
```

Our unscramble code is similar. There are just a few lines that must be changed:

```
encrypt_zip(2,i) =encrypt_zip(1,i) ;  -TO-
        encrypt_zip(2,i) =encrypt_zip(1,j) ;

put 'first loop: start element in position ' i ' is now ' encrypt_zip(1,i);  -TO-
        put 'first loop: start element in position ' j ' is now '  encrypt_zip(1,j);

put 'first loop: target element in position ' j ' is now '  encrypt_zip(2,j); -TO-
        put 'first loop: target element in position ' i ' is now ' encrypt_zip(2,i);

put 'second loop: start element in position ' i ' is now ' encrypt_zip(1,i); -TO-
        put 'second loop: start element in position ' j ' is now '  encrypt_zip(1,j);

put 'second loop: target element in position ' j ' is now '  encrypt_zip(2,j); -TO-
        put 'second loop: target element in position ' i ' is now ' encrypt_zip(2,i);
```

There will be a new variable, zip_orig to hold the unscrambled code so that we can test this program against the original values. See appendix for complete code.

Our log for the unscramble is:

```
array at start 0 0 1 5 6 X X X X X
first loop: start element in position 3  is now 1
first loop: target element in position 2  is now 1
first loop - array after 2  and 3  processed 0 0 1 5 6 X 1 X X X
first loop: zip_orig is 1
first loop: start element in position 4  is now 5
first loop: target element in position 3  is now 5
first loop - array after 3  and 4  processed 0 0 1 5 6 X 1 5 X X
first loop: zip_orig is 15
first loop: start element in position 5  is now 6
first loop: target element in position 4  is now 6
first loop - array after 4  and 5  processed 0 0 1 5 6 X 1 5 6 X
first loop: zip_orig is 156
first loop: start element in position 1  is now 0
first loop: target element in position 5  is now 0
first loop - array after 5  and 1  processed 0 0 1 5 6 X 1 5 6 0
first loop: zip_orig is 1560
stop second loop at 1
second loop: start element in position 2  is now 0
second loop: target element in position 1  is now 0
second loop - array after 1  and 2  processed 0 0 1 5 6 0 1 5 6 0
second loop: zip_orig is 01560
```

Now, we are left with what is really too much information in the scrambled output. Although the logic is fairly simple, why give any more clues to it than necessary? We should remove everything except out_zip:

CURRENT_ZIP 1	CURRENT_ZIP 2	CURRENT_ZIP 3	CURRENT_ZIP 4	CURRENT_ZIP 5	NEW_ZIP 1	NEW_ZIP 2	NEW_ZIP 3	NEW_ZIP 4	NEW_ZIP 5	SWAP1	SWAP2	I	J	STOPLOOP2	IN_ZIP	OUT_ZIP
0	1	5	6	0	0	0	1	5	6	2	3	3	4	1	01560	00156

Illustration 11

We can fix by dropping I, j, stoploop2, and in_zip from the output. Instead of creating the two data set variables SWAP1 and SWAP2, we'll just use the macro variables PARM1 and PARM2 directly. Better than that, we can use a temporary array so that none of the current_zip and new_zip variables are retained in the output:

```
data testscramble(drop = i j stoploop2 in_zip);
array encrypt_zip(2,5) $1. _temporary_;
attrib zip_out format=$5.;
select (&PARM1);
        when (1) STOPLOOP2=0;
        when (2) STOPLOOP2=1;
        when (3) STOPLOOP2=2;
        when (4) STOPLOOP2=3;
        when (5) STOPLOOP2=4;
end;
…..
I=&PARM1;
J=&PARM2;
```

See appendix for entire code.

CONCLUSION

We've seen some uses of arrays and how to test that they are being correctly accessed and populated. We've also seen how to hold and drop your array with the use of the _TEMPORARY_ variable. Besides scrambling values and working with the output from a proc transpose, arrays are also useful for those cases when you need more control over pivoting your columns than proc transpose can give.

ACKNOWLEDGEMENTS

Thank you, Joe Butkovich for reviewing this paper and presentation

RECOMMENDED READING
SAS Documentation Online 9.4 – SAS Institute

CONTACT INFORMATION

Your comments, questions and experiences are valued and encouraged. Contact the author at:

Patricia Hettinger
Oakbrook Terrace, IL 60523
Phone: 331-462-2142, cell 630-309-3431
Email: patricia_hettinger@att.net

APPENDIX

Entire unscramble program, test version

```
data testunscramble;
array encrypt_zip(2,5) $1. current_zip1-current_zip5
new_zip1 - new_zip5;
attrib zip_orig format=$5.;
SWAP1 = &PARM1;
SWAP2=&PARM2;
select (SWAP1);
        when (1) STOPLOOP2=0;
        when (2) STOPLOOP2=1;
        when (3) STOPLOOP2=2;
        when (4) STOPLOOP2=3;
        when (5) STOPLOOP2=4;
end;
set TESTSCRAMBLE;
do i = 1 to 2;
        do j = 1 to 5;
        if i = 1 then
        encrypt_zip(i,j) = substr(zip_out,j,1);
        else encrypt_zip(i,j) = 'X';
        end;
end;
put 'array at start ' encrypt_zip(*);
I=SWAP1;
J=SWAP2;
/*Only difference between scramble and unscramble is the subscripts are reversed*/
do until (I gt 5 );
        encrypt_zip(2,i) =encrypt_zip(1,j) ;

        put 'first loop: start element in position ' j ' is now '  encrypt_zip(1,j);
        put 'first loop: target element in position ' i ' is now ' encrypt_zip(2,i);
        put 'first loop - array after ' i ' and ' j ' processed ' encrypt_zip(*);
        substr(zip_orig,i,1)=encrypt_zip(2,i);
        put 'first loop: zip_orig is ' zip_orig;
        I+1;
        J+1;
        if J gt 5 then J = 1;
end;
if SWAP1 gt 1 then do;
if I gt 5 then I = 1;
if J gt 5 then J = 1;
        do until (I gt STOPLOOP2);
        encrypt_zip(2,i) =encrypt_zip(1,j) ;
        put 'stop second loop at ' stoploop2;
        put 'second loop: start element in position ' j ' is now '  encrypt_zip(1,j);
        put 'second loop: target element in position ' i ' is now ' encrypt_zip(2,i);
        put 'second loop - array after ' i ' and ' j ' processed ' encrypt_zip(*);
        substr(zip_orig,i,1)=encrypt_zip(2,i);
        put 'second loop: zip_orig is ' zip_orig;
        I+1;
        J+1;
        if J gt 5 then J = 1;
        end;
end;
run;
```

Entire scramble program – 'production' version in which all calculation variables are dropped.

```
data testscramble(drop = i j stoploop2 in_zip);
array encrypt_zip(2,5) $1. _temporary_;
attrib zip_out format=$5.;
select (&PARM1);
        when (1) STOPLOOP2=0;
        when (2) STOPLOOP2=1;
        when (3) STOPLOOP2=2;
        when (4) STOPLOOP2=3;
        when (5) STOPLOOP2=4;
end;
/*initialization */
do i = 1 to 2;
        do j = 1 to 5;
        if i = 1 then
        encrypt_zip(i,j) = substr(zip_in,j,1);
        else encrypt_zip(i,j) = 'X';
        end;
end;
I=&PARM1;
J=&PARM2;
/*first loop will execute until we have finished processing start element 5*/
/*if parm1 equal to 1, this is the only loop that will execute*/
do until (I gt 5 );
        encrypt_zip(2,j) =encrypt_zip(1,i) ;
        substr(zip_out,j,1)=encrypt_zip(2,j);
        I+1;
        J+1;
        if J gt 5 then J = 1;/*need this so target array doesn't go out of bounds*/
end;
/*if parm1 is greater than 1, this loop will execute too - it will stop at*/
/*the predetermined stoploop2*/
if &PARM1 gt 1 then do;
        if I gt 5 then I = 1;
        if J gt 5 then J = 1;
        do until (I gt STOPLOOP2);
                encrypt_zip(2,j) =encrypt_zip(1,i) ;
                substr(zip_out,j,1)=encrypt_zip(2,j);
                I+1;
                J+1;
                if J gt 5 then J = 1;
        end;
end;
run;
```

- The authors of the paper/presentation have prepared these works in the scope of their employment with Patricia Hettinger, Independent Consultant and the copyrights to these works are held by Patricia Hettinger.
  Therefore, Patricia Hettinger hereby grants to SCSUG Inc a non-exclusive right in the copyright of the work to the SCSUG Inc to publish the work in the Publication, in all media, effective if and when the work is accepted for publication by SCSUG Inc.
  This the 02 day of October, 2017.