

# Doing More with SAS® Arrays

## Rahul G. Pillay, Colchis Capital, San Francisco, CA

Have you used SAS® Arrays to perform repetitive calculations or create multiple variables? Maybe you find yourself understanding a bit about SAS® Arrays but would like to do more. This paper will help us explore Arrays a bit further. Do you need to transpose data and do additional data manipulations and calculations in the process? Have you considered the possibilities of using two-dimensional Arrays? We will explore examples of transposing and manipulating data, performing complex calculations and briefly explore two-dimensional Arrays.

### Introduction

SAS® Arrays are a group of variables grouped together for the duration of a Data step. We pass by Arrays every time we go to the grocery store. There are a group of cars parked in parking spots outside – the lanes organized in rows are Arrays of cars. The isles with the produce can have an Array of vegetables and an array of fruits. SAS® allows us to reference a group of like variables by creating and manipulating them in Arrays. An example would be to move all the vegetables from Aisle 1 to Aisle 7. Ideally you can define an Array that contains all the vegetables in Aisle 1 and move them to Aisle 7 by referencing to the defined Array.

There are several very good papers on SAS® Arrays. For a quick introduction I would recommend reading Ron Cody's *SAS® Functions by Example* and searching for keyword Arrays. There are many papers that authors wrote for several conferences including *Quick Tips on Powerful Use of SAS® Arrays*, a paper I wrote for WUSS 2014.

Arrays are a set of variables grouped together for the duration of a Data step – these could be the vegetables we described above. If you were to temporarily display our vegetables or SAS® variables in each column of a spreadsheet they would look like **Table 1a**. Any combination of the vegetables, specifically the columns in our spreadsheet can be referenced in a SAS® Array.

**Table 1a** shows five columns from a 'spreadsheet of variables'. This is an example of an array of vegetables.

vegetable1	vegetable2	vegetable3	vegetable4	vegetable5
Good Celery	Ripe lemon	Good Cucumber		Small tomato
Bad celery	Green lemon	Bad Cucumber	Rotten potato	Large tomato
.	.	.	.	.
.	.	.	.	.

**Table1a. Spreadsheet with 5 columns**

We can group multiple columns (or vegetables) temporarily in an Array and reference them with an Array statement at any point during the Data step. For example, let's extract the second word from every vegetable listed by creating a new variable called vegetable\_new. If we performed this calculation using 'if then' statements, we would need to repeat the same calculation for each variable as shown in *Example 1a* below.

#### Example 1a

```
data ex1a;          /*sample data set*/
set Tbl1a;
    vegetable_new1 = scan(vegetable1,2);      /*creating vegetable_new1*/
    vegetable_new2 = scan(vegetable2,2);      /*creating vegetable_new2*/
    .
    .
    vegetable_new5 = scan(vegetable5,2);      /*creating vegetable_new5*/

run;
```

In essence we are trying to accomplish the task visually shown in **Table 1b** where we have five additional variables with new vegetable names.

**Table 1b** displays 5 new columns with variables vegetable\_new1, vegetable\_new2, and so forth.

vegetable1	vegetable2	vegetable3	vegetable4	vegetable5	vegetable_new1	vegetable_new2	.	.	.
Good Celery	Ripe lemon	Good Cucumber		Small tomato	Celery	lemon	Cucumber		tomato
Bad celery	Green lemon	Bad Cucumber	Rotten potato	Large tomato	Celery	Lemon	Cucumber	Potato	tomato
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

**Table1b. Spreadsheet with 10 columns listing 5 new columns**

An alternative to performing *Example 1a* would be to use an Array statement in a DO LOOP. We know that **Table 1a** lists a group of variables that can be referenced with an Array statement. All we need is another Array statement to reference the additional group of variables (new vegetable variables) needed to be created as shown in **Table 1b**. Consider *Example 1b*.

**Example 1b**

```
data Ex1b;
    set tbl1a;
    array vegetablearray_old {5} $ vegetable1 - vegetable5;
        /*using array for old vegetables*/
    array vegetablearray_new {5} $ vegetable_new1 - vegetable_new5; /
        /*creating new variables*/
    do i=1 to 5;
        vegetablearray_new {i} = scan(vegetablearray_old {i},2);
        /*vegetable_new = scan(vegetable_old,2)*/
    end;
run;
```

You can imagine how effective this can be especially when dealing with hundreds of variables. In *example 1b* we discussed a consecutive list of elements (**vegetable1 – vegetable5, vegetable\_new1 – vegetable\_new5**) but this isn't necessary for all Array statements. The array below 'Array\_example' lists a random set of elements.

r refers to the first element, a to the second, h to the third and so forth.

*Example:*       ARRAY Array\_example {5} r a h u l;

## SYNTAX

An array is defined with an ARRAY statement.

```
ARRAY array_name {n} <$> <length> array-elements <initial values>;
```

*array\_name*: a valid SAS® name that is not a variable name in the data set.

*{n}* : number of elements in the array

*<\$>* : indicates if the elements within the array are character or numeric. For character elements include the (\$) sign in the syntax. If the dollar sign is not included, the array is assumed to be numeric.

*<length>* : a common length for the array elements.

*<initial values>*: any initial values to be assigned to each array element.

Arrays only exist for the duration of a data step so it is necessary to define arrays before referencing them in any Data step. Array statements cannot be used in DROP or KEEP statements.

An Array can either contain all numeric elements or all character variables but not both. A quick and easy way to reference all character, numeric or all the variables would be to use the following syntax:

*\_numeric\_*       all numeric variables will be used in the data step.

*Example:*       ARRAY Array\_example {5} \_numeric\_;

*\_character\_*     all character variables will be used in the data step.

*Example:*       ARRAY Array\_example {5} \_character\_;

`_all_` all the variables will be used if they are the same type.  
**Example:** `ARRAY Array_example {5} _all_;`

It is not necessary to list the elements of an Array while creating new variables. The array below will create the variables `Array_example1`, `Array_example2` and `Array_example3`.

**Example:** `ARRAY Array_example {3};`

Since the dollar symbol (\$) is missing and the Array statement specifically states that the array has three elements SAS® automatically assigns these numeric variables a name.

We don't have to specifically specify the number of variables in an array. The star (\*) symbol will tell SAS® to determine the number of elements in as Array. This can be useful in instances where you aren't sure of the number of variables being specified for the array in any given data set.

**Example:** `ARRAY Array_example {*} _character_;`

## Transposing Data

Now let's do more with SAS® Arrays. Sometimes datasets are not necessarily in the shape or form that they need to be to carry out the analyses needed. A tall dataset may need to be wide or a wide dataset may need to be tall, observations may need to be variables or variables may need to be observations. In this section we will explore how to transpose data using SAS® Arrays. Many SAS® users prefer to use PROC TRANSPOSE rather than SAS® Arrays which can offer simpler syntax but Arrays can offer solutions to more complex problems such as transposing and manipulating data. Some examples are as follows: -

- Arrays can be handy when naming variables whereas PROC TRANSPOSE requires the use of special options such as `id`, `prefix` or `suffix`.
- Arrays can be used for many other purposes such as manipulating variables, performing table lookups, creating new variables. If you want more options in restructuring your data Arrays can be extremely helpful.

**Table 2a** details the loan payments of members by each month for first quarter. Our goal is to transform this dataset so that we have 1 observation per member and each payment as a variable. Please note that this dataset is already sorted by member, which is critical for all the examples described in this paper to work properly. You may have to do this for your use case as well.

Member	Month	Payment
1	Jan	\$233.44
1	Feb	\$235.44
1	Mar	\$239.87
2	Jan	\$411.88
2	Feb	\$411.88
2	Mar	\$425.88
3		
.		
.		

## Table 2a. Sorted Table with data for Example 2

### Example 2a:

```
Proc transpose data=tbl2a out=ex2a (drop=_name_);  
By member; var Payment;  
Prefix = Month;  
Run;
```

As mentioned earlier PROC TRANSPOSE offers simpler syntax when only transposing is needed. The Prefix= options tells SAS® to start naming the transposed columns with a Prefix of Month – the columns will be labeled Month1, Month2, etc. The default in SAS® labels them as COL1, COL2, etc.

With arrays you can achieve this by *example 2b*:

**Example 2b:**

```
data ex2b (drop=i day payment); /*create dataset y - drop unwanted variables*/

set x; by member;              /*since we want to make sure that we obtain 1
                                obs per member*/
array monthpayment{3} Jan Feb Mar;
                                /*Create a numeric Array with 3 variables -
                                Jan, Feb, Mar*/
if first.member then i = 1;     /*initialize i for every first occurrence of
                                member*/
    monthpayment{i} = payment;   /*create column for each
                                payment*/
    if last.member then output;  /*output when last occurrence of
                                member*/
    i+1;                        /*increase I value for next
                                iteration */

retain Jan Feb Mar;
run;
```

It is obvious that the DATA step requires more syntax and a better understanding of how SAS® transposes the data. One major issue to note in this example is that each member has the same number of payments made for each month or that there are no missing values. Unfortunately this is not the case when working with datasets at work. There are ways to deal with this scenario using the DATA step but PROC TRANSPOSE would make it simpler. In the following examples we will discover ways in which we can handle datasets that we are more familiar with; where payments are missing or each member may not have the same number of observations.

Consider **Table 2b** where Member 4 doesn't have a Feb payment and March payment is missing. Will the Array work in this case?

Member	Month	Payment
1	Jan	\$233.44
1	Feb	\$235.44
1	Mar	\$239.87
2	Jan	\$411.88
2	Feb	.
2	Mar	\$425.88
3	Jan	\$599.09
3	Feb	\$599.09
3	Mar	\$549.09
4	Jan	\$12.64
4	Mar	.

**Table2b. Table showing missing Payments and observations for Members 2 and 4.**

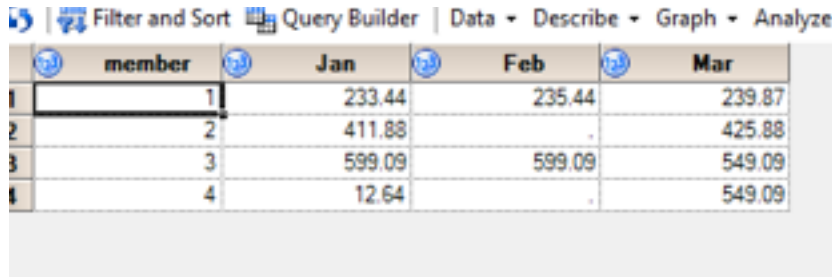
Before we answer that consider **Table 2c** where we visualize the result dataset:

Member	Jan	Feb	Mar	Total
1	\$233.44	\$235.44	\$239.87	\$708.75
2	\$411.88	.	\$425.88	\$837.76
3	\$599.09	\$599.09	\$549.09	\$1747.27
4	\$12.64	.	.	\$12.64

**Table2c. The goal of our results for Example 2.**

Unfortunately if we used the code described in *Example 4b* we won't quite achieve our goal because the March payment for Member 4 is still the retained amount from Member 3. The output for this is displayed below. Note that the March payment for Member 4 comes from Member 3.

**Table 2d.** shows the SAS® Output if *Example 4b* was used to transpose **Table 2c**.



	member	Jan	Feb	Mar
1	1	233.44	235.44	239.87
2	2	411.88	.	425.88
3	3	599.09	599.09	549.09
4	4	12.64	.	549.09

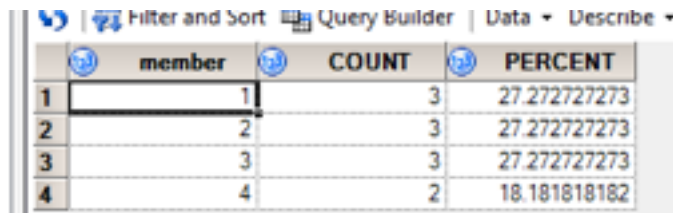
**Table 2d.** The incorrect results when using *Example 2b* (Member 4 March payment) with missing observations or variables.

There are a few ways to handle this but one clever way would be to know what the maximum number of observations that exist across all the members in the dataset. This way we can arrange our logic to account for any number of observations for each existing member in our dataset and create a DO LOOP specific to each member. This means that Members 1, 2 and 3 will loop 3 times while Member 4 only loops once. Let's work step by step. First we will figure out the number of observations for each member in our dataset via PROC FREQ.

#### **Example 2c Part1:**

```
Proc Freq Data=tbl2a order=freq;                /*order table by frequency*/
    tables member / noprint out=temp;           /*create dataset without print*/
run;
```

**Table 2e.** shows the SAS® Output of *Example 2c*.



	member	COUNT	PERCENT
1	1	3	27.272727273
2	2	3	27.272727273
3	3	3	27.272727273
4	4	2	18.181818182

**Table 2e.** Results of *Example 2c Part 1*.

**Table 2e** shows the temporary dataset called TEMP ordered by descending Counts of observations. Option `order=freq` helps us achieve this easily so that we can create a macro variable from the first observation of TEMP dataset.

#### **Example 2c Part2:**

```
Proc Sql;
    Select Count into: n                /*Create macro variable for maximum
    from temp;                          count*/
quit;
```

**Example 2c Part2** helps us create the macro variable `n` with the use of the `into:` statement in PROC SQL. This variable is the maximum Count for number of transactions per member from our original dataset. In this case `&n` will resolve to 3.

```
data ex2c (drop=i payment month);
    array monthpayment{&n}            /*Create Array monthly payment*/
    do i =1 to &n until(last.member);
    /*do loop for a maximum of 3 or until the last occurrence of Member*/
```

```

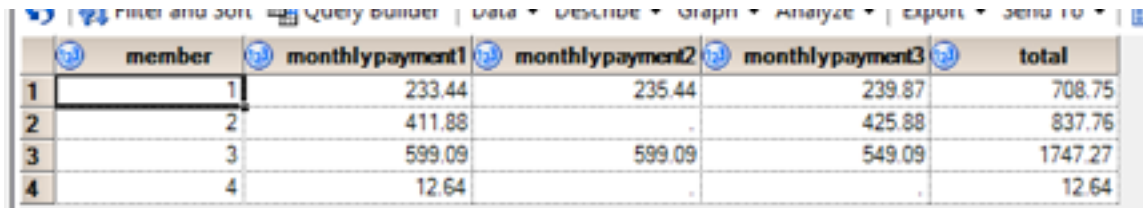
set tbl2a ;                                /*set statement within do loop*/
by member;                                /*set by member*/
  monthlpayment{i} = payment;              /*create column for each payment*/
  total = sum(total,payment);              /*calculate total of payments*/
  if last.member then output;              /*1 observation per member*/
end;

run;

```

The Array statement in **Example 2c** is much simpler than what we have discussed before. In this case we let SAS® create the new variables for us by determining the number of variables specified when &n resolves to 3. This would look something like monthlpayment1, monthlpayment2, and monthlpayment3. In the first instance when i=1 our SET statement will grab the first member (A SET statement within a DO LOOP helps perform the calculations and other logic while the last observation is reached avoiding the need for retain statements) and Monthlpayment1 = \$233.44. The sum function statement helps us initialize and calculate the total variable that we wanted to calculate for all the payments made by a particular member. Since this is not the last observation for Member 1 we return to the top of the loop and reiterate with i=2. This keeps happening as we create new columns for monthly payments and adding them to the total variable until we get to the last payment for each of the members – the output we want. The key difference that this DO LOOP provides versus **Example 2b** is when we get to Member 4. When i=2 the until(last.member) helps us finalize the iteration versus rerunning for i=3. This way we won't have any prior values retained from prior members in the data vector when creating the output. The DO LOOP ends and gives us the results we want:

**Table 2f. Final Results for Example 2c.**



	member	monthlpayment1	monthlpayment2	monthlpayment3	total
1	1	233.44	235.44	239.87	708.75
2	2	411.88	.	425.88	837.76
3	3	599.09	599.09	549.09	1747.27
4	4	12.64	.	.	12.64

**Table 2f.** The correct results for **Example 2c**.

Another clever way to handle missing variables would be to create a multi-dimensional Array holding each of the Month and Payment for that month. In this case we would only need a 2 dimensional Array since we have 2 variables to consider: Member and Payment. As the number of variables increases – so does the Array dimensions. A two-dimensional Array can be identified as ARRAY\_NAME {4,3} where 4 = 4 rows and 3 = 3 columns.

In **Example 3** we will transpose **Table 2f** above to what our original dataset looked like in **Table 2b**.

```

data ex3 (keep=Member1 monthlpayment1 monthlpayment2 monthlpayment3);
  set tbl2f end=last;                                /*create indicator for end of dataset*/
  array monthlpayment{3} monthlpayment1 monthlpayment2 monthlpayment3;
                                                    /*array to hold payments*/
  array all{4,3} _temporary_;                      /*2 dimensional array with 4 rows and 3
                                                    columns*/
  array member_new{4}                                /*array for members*/
  i + 1;
    do j = 1 to dim(member_new);
      all{i,j} = monthlpayment[j];                  /*creating payment columns
      within loop for each member*/
    end;
  if last then do;                                    /*creating final dataset when end of
                                                    dataset is reached*/
    do j = 1 to dim(monthlpayment);
      do i = 1 to 4;
        member_new[i] = all[i,j];
      end;
    end;
  output;

```

```
end;  
run;
```

**Example 3** looks complicated but can be powerful once understood. The key here is to create an array that can hold all the variables that we want transposed and then assigning them to their corresponding members in the final step. `array monthlypayment{3}` creates an array with monthly payments. `array all{4,3} _temporary_` creates a temporary 2X3 Array with 4 rows and 3 columns. This is for 4 members and at most 3 payments. `array member_new` is an array that will help us assign the 4 members to their corresponding payments. When `i=1` we start a do loop which will iterate until it ends. When `j=1` the Array `all{1,1}` = \$233.44, When `j=2` Array `all{1,2}` = \$235.44 and when `j=3` Array `all {1,3}` = \$239.87. This iteration continues for `i=2, 3` and `4` until we have an array that looks like **Table 3a** in the program data vector (PDV).

**Table 3a** shows the all Array from **Example 3**.

1	\$233.44	\$235.44	\$239.87
2	\$411.88	.	\$425.88
3	\$599.09	\$599.09	\$549.09
4	\$12.64	.	.

**Table 3a** showing the Array All in the PDV.

After the final iteration the `if last then do` statement helps us trigger the final logic since we are at the end of the dataset. A new do loop, `do j = 1 to dim(monthlypayment)`, helps s create a do loop for each of the payments in the all array which the nested do loop, `do i = 1 to 4; member_new[i] = all[i,j]` helps assign each of the Members to their corresponding payments resulting in the final dataset.

### Conclusion

We learned how SAS Arrays can handle simple manipulations, like referencing a set of variables and renaming them to complex transformations, like transposing datasets and calculating totals. There are other ways to handle similar tasks but a clear understanding of how Arrays work will help add another technique to your SAS® knowledge. After all SAS® programmers are constantly trying to find a better and more efficient way of solving problems and SAS® Arrays helps us do just that in our DATA step.

## References

Burlew, Michele M. *SAS® Macro Programming Made Easy*. 2nd ed. Cary, NC: SAS® Institute, 2006. Print.

Carpenter, Art. *Carpenter's Complete Guide to the SAS® Macro Language*. 2nd ed. Cary, NC: SAS® Institute, 2004. Print.

Cody, Ronald P. *Learning SAS® by Example a Programmer's Guide*. Cary, N.C.: SAS® Institute, 2007. Print.

Cody, Ronald P. *SAS® Functions by Example*. 2nd ed. Cary, N.C.: SAS® Institute, 2010. Print.

## Acknowledgements

*I wish to express my sincere thanks to Jon Cooper for all his support, encouragement and guidance in writing this paper, Karen Ran Bi, Melissa Cliatt and Kevin McGlynn for their training and mentorship in credit, and Robert and Edward Conrads for adding countless value into my work and career.*

## Contact Information:

**Name:** Rahul G. Pillay  
**Enterprise:** Colchis Capital Management  
**Address:** 1 Maritime Plaza # 1975  
**City, State ZIP:** San Francisco, CA  
**Work Phone:** (415) 400-8612  
**E-mail:** rahul@colchiscapital.com  
**Web:** <http://www.colchiscapital.com/>

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.