Off and Running with Arrays in SAS®

Stephen Keelan, SAS Canada, Toronto, On

ABSTRACT

Often we look at our newly completed SAS® program and think … there must be a better way. If your programs have many repetitious lines of code, calculating and re-calculating the same thing, or if you need to rotate or transpose data … Arrays may be for you. This tutorial will focus on introducing you to Arrays, understanding what they are and how to add them to your SAS programs. This tutorial will use Base SAS® and is appropriate for beginner and intermediate SAS programmers.

INTRODUCTION

In SAS one of the most powerful transformation engines is the Data Step. Part of it's power is in the flexibility it affords in different approaches that can be used to solve a business problem. With arrays you can reduce or simplify the coding in many cases and in other cases accomplish tasks not easily done other wise. Starting with the basics and building from there we will see what Arrays are and how we can benefit from using them, either through modifying our existing programs or at least in starting to use them in new applications that you develop.

GETTING STARTED WITH ARRAYS

In developing an application or simply writing some code to create the data or the required report, one approach can be to write the code first then look to improve it later. Step one would mean writing out repetitious lines of code, following the logic and the business rules to create the desired results. Then, having understood the logic needed and recognized the repetition, step two would be to use arrays to reduce the redundant code. Alternatively, with experience in using arrays, you could jump to writing the desired code with Arrays to begin with.

THE WHAT AND THE WHY

What is an array? Why use an Array? Well, at a high level an array provides you with a means of dynamically referring to a group of variables, through what are called an array reference and a subscript. If in your process, you have the same calculation or transformation that needs to be done on several variables, using an array and just as important, how the array is processed can reduce the code required.

If we don't worry about repetitious code just yet, here is some Retail data where we have several variables that have the price of different products that are sold in various locations across the country. Our business need is to apply a discount of 10% to all products (all product variables) within each location (each observation). To code this we might include an assignment statement for each product variable:

```
data discount;
     set CurrentPrice(keep=location
                          prod1
                          prod2
                          prod3
                          prod4);
     Prod1=Prod1*(1-.1);
     Prod2=Prod2*(1-.1);
     Prod3=Prod3*(1-.1);
     Prod4=Prod4*(1-.1);
run;
```

For this application, we have applied the discount and if we do only have 4 products this program is most likely satisfactory. If we have hundreds of products that's when this code becomes extremely repetitious and where Arrays can help us out greatly.

THE SYNTAX

A SAS array is a collection of SAS variables that can be referenced in the Data Step under a common, single name. The general syntax for defining an array is as follows:

ARRAY array-name{dimension} $ length elements (initial values);

- ARRAY – is the Identifying Keyword for the statement.
- Array-name – is the name we create for the array. It must be a valid SAS name and is recommended to not be the same as a SAS Function name. In Version 7 and beyond the array name can be up to 32 characters in length.
- {Dimension} – indicates the number of elements (or variables referenced) in this array.
- $ - included on the ARRAY statement only if the array is character, that is, if the array will be referencing new character variables.
- Length – can be used to define the length of the new character variables referenced by the array.
- Elements – can be used to define the variables that the array will reference, either existing variables or new variables.
- Initial Values – can be included to give the elements of the array initial values. This also causes these variables to be retained during the data step (i.e. not reinitialized to missing at the execution of the DATA statement).

Before looking at additional rules and recommendations, here's an example of defining an array to help with our discount calculations:

```
data discount;
     set CurrentPrice;
     array Products {4} prod1-prod4;
…
```

At this point in the program, the first part of our work is done in that we have defined an array. This is a must before we move to the second part where we will "process" the array. On the ARRAY statement, the array-name is Products and it has a dimension of 4 meaning it will reference four numeric variables in this data step. The elements or variables it will reference have been defined on the ARRAY statement as prod1, prod2, prod3 and prod4 using a single hyphen to build the implied list. By specifying the elements on the ARRAY statement and placing it after the SET statement, the Products Array will reference the Prod1-Prod4 variables that are brought in from the CurrentPrice data set by the SET statement. In this case, the prod variables are all numeric with a default length of 8 bytes.

To move on to the second part of "processing" the array, using a simple DO loop will accomplish the repetitive calculation with just one occurrence of the assignment statement. Notice from the our first attempt at solving this with the 4 assignment statements, the only part that is changing is the number of the Prod variable

that is being used in the calculation. The array allows us to dynamically set which element of the array (and therefore which variable) we are referring to as follows:

```
data discount;
     set CurrentPrice(keep=location
                            prod1
                            prod2
                            prod3
                            prod4);
     array Products {4} prod1-prod4;
     do j = 1 to 4;
            Products{j}=Products{j}*(1-.1);
     end;
run;
```

Now we have reduced the … well, given that we only had four assignment statements to start with, we actually have ended up with the same number of statements. However, the combination of the ARRAY statement and the DO loop has enabled us to process a group of variables, whether it's 4 or 400, with a single assignment statement.

MORE DETAILS

The ARRAY statement is a Compile time only statement meaning that it is not executed during the execution of the program, it is only considered during the compile phase. The Array and the ability to reference elements using the array name and a subscript value are only valid for the duration of the Data Step. Subsequent Procedure steps can only reference variable names, not the array name. The same applies for subsequent Data Steps, however you could simply redefine the array in this Data Step to do further processing of the group of variables using the new array. Variable names must also be used on LABEL, FORMAT, LENGTH, DROP or KEEP statements, not the array reference.

In the Data Step, the order of the statements is important in both the compile and execution phases and this holds true for the ARRAY statement in the compile phase. The ARRAY statement must be defined in the Data Step prior to any references to the array in other Data Step statements. If the elements are not specified on the ARRAY statement, SAS will use the Array name, append an element number as a suffix starting at 1 and check to see if that variable name exists already in the Program Data Vector (PDV). If those variable names do not exist, it is the array that actually creates them as variables in the PDV. So, for our example given that the input data set contains the variables Prod1-Prod4 and our array name is Products, if we simply left off the elements from the ARRAY statement, our program would not work as we had hoped:

```
data discount;
     set CurrentPrice(keep=location
                            prod1
                            prod2
                            prod3
                            prod4);
     array Products {4};
     do j = 1 to 4;
            Products{j}=Products{j}*(1-.1);
     end;
run;
```

In this program during compile the SET statement would add the variables specified on the KEEP= data set option to the PDV then the ARRAY statement would be encountered. Since there are no elements defined, SAS will check for and then add Products1, Products2, Products3 and Products4 to the PDV. The end result will be that we have 4 additional variables added to the Discount data set all with missing values instead of the discounted price as intended. One way to make this program work is to change the array name to align with the variable names, though this can get confusing keeping track of what is a variable name and what is a reference to the array.

```
data discount;
     set CurrentPrice(keep=location
                            prod1
                            prod2
                            prod3
                            prod4);
     array Prod {4};
     do j = 1 to 4;
            Prod{j}=Prod{j}*(1-.1);
     end;
run;
```

Some additional points to keep in mind for Arrays are that they can only be Character or Numeric, not a combination of both. When specifying the dimension of the array you don't have to use curly braces {4} or for the index variable used {j} when processing the array. Regular brackets (x) are acceptable syntax but the curly braces help to distinguish an array reference for you and your colleagues reading or updating the program. For the array name, it can be the same as a SAS Function but it is recommended that you avoid this as your program will lose the use of that function. Here is a short example of this with the warning from the LOG.

```
data test;
     array sum {2} (10,20);
     x=sum{1};
     y=sum{2};
run;
```

```
WARNING: An array is being defined with the same
name as a SAS-supplied or user-defined function.
Parenthesized references involving this name
will be treated as array references and not
function references
```

The program does in fact run, the data set test has 4 variables: sum1, sum2, x and y but if we tried to use the sum function it would be considered as an array reference.

To this point we have developed a simple example to demonstrate the syntax and concepts of an array in SAS. In general we now have a way of processing a group of numeric variables all in the same way.

MORE FEATURES

In this section we will expand the use of arrays to include lists of variables that don't have a numeric suffix and also define a Character Array. Arrays can also be used to create a group of new variables with the same attributes for type and length.

In our retail data, rather than having generic product variable names like Prod1-Prod4, we may have more descriptive variable names that don't end in a convenient numerical suffix. In this case you need to specify each of the variables on the ARRAY statement, in the order you wish to process them.

```
data discount;
     set ProductNames(keep=location
                            radio
                            TV
                            microwave
                            toaster);
```

```
     array Products {4}     radio
                            TV
                            microwave
                            toaster;
     do j = 1 to 4;
            Products{j}=Products{j}*(1-.1);
     end;
run;
```

This example helps emphasize that the array name and the variable names don't need to be the same or even similar and don't need to have a numerical suffix. By specifying the elements on the ARRAY statement (radio, TV etc) we are aligning the 4 array references to the variables coming in from the input data set. Otherwise, without specifying the elements, we would end up with 4 new variables (Producst1, Products2 etc.) that the array would reference. When we are processing the array we can use the same DO loop as we first did as we don't have to worry about substituting the variable names into the code (Radio = Radio * …) that's what the array reference and subscript do for us. One additional point about the subscript, it can also be an expression so, for example we could use a mathematical operator to allow us to shift the element being referenced up or down. For example, to calculate differences between months i.e. Feb – Jan and Mar – Feb etc. you could use the following where each observation is one year of historical data :

```
data Compare;
     set yearly;
     array monthly{12} Jan Feb Mar
                        Apr May Jun
                       Jul Aug Sep
                       Oct Nov Dec;
     Array difference{11};

do k=1 to 11;
     difference{k}=monthly{k+1} - monthly{k};
end;
run;
```

In this example, we would now have 11 new variables that would contain the difference between the months for each year.

CREATING NEW VARIABLES

There may also be a need to create a new group of variables for different reporting needs. Rather than modify the value of the original Product variable, create a new variable to hold each of the new prices, so we can maintain the original price and perhaps do some comparison calculations. The long way to write this code would be as follows:

```
NewPrice1=radio*(1-.1);
NewPrice2=TV*(1-.1);
NewPrice3=microwave*(1-.1);
NewPrice4=toaster*(1-.1);
```

To handle this type of situation with arrays, we will define a second array to create the new variables and include this array reference in the do loop as follows:

```
data discount;
     set ProductNames(keep=location
                            radio
                            TV
                            microwave
                            toaster);
     array Products {4}     radio
                            TV
                            microwave
                            toaster;
     array NewPrice {4};
     do j = 1 to 4;
            NewPrice{j}=Products{j}*(1-.1);
     end;
run;
```

As mentioned earlier, for the NewPrice Array there are no elements specified on the ARRAY statement so SAS will create 4 new variables (Numeric, length of 8) that will contain the new discounted price of the products.

AUTOMATING TECHNIQUES

One of the themes of this paper is to reduce redundant code, following closely with that theme is the desire to have code that is easily maintained, especially if the data that we are working with changes. In our Retail example, new products are always being invented and therefore added to store inventories, and some products that were not so successful get dropped. In our program so far, the array and the do loop will handle hundreds of products with the one assignment statement but there are a few limitations. One limitation is that as new products get added and deleted from our input data set we will have to update the elements listed on the ARRAY statement. Another maintenance task will be updating the dimension of our array and the corresponding STOP value on the do loop. If the STOP value in a DO Loop that is processing an array is greater than the dimension of any of the arrays referenced in that DO Loop, the Data Step will terminate with an Error in the log. For example:

```
array NewPrice {4};
     do j = 1 to 5;
            NewPrice{j}=Products{j}*(1-.1);
     end;

ERROR: Array subscript out of range …
```

In order to automate the definition of the array to make our code easier to maintain, we can use a key word and a function to have SAS populate the variables that will be referenced in the array and count how many that will be. This will be dependant on the structure of the incoming data set, in our example we have only one character variable for the Location and the rest are all Numeric variables that contain a price for a product. So, instead of listing the elements on the ARRAY statement we will use the key word _NUMERIC_. When the ARRAY Statement is compiled, using _NUMERIC_ will take all of the numeric variables in the PDV at that time and define them in order to be the elements of the array. Since our input data may change quite frequently we won't always know, or want to know how many Product variables there will be so how would you code the dimension of the Array? SAS allows an asterisk {*} to be specified as the dimension of the Array if you wish SAS to calculate the number of elements. To calculate the dimension, SAS would count either the number of elements specified as elements on the ARRAY statement or by counting how many it found in the PDV using _NUMERIC_. The last part then of reducing the maintenance of our program is to transfer this SAS calculated dimension of the array to the STOP value of the DO Loop where we will be processing the array. In the Data Step, you can use the DIM function for this and putting all the pieces together would look like the following:
Note: in this program the KEEP= data set option has been dropped, it's main purpose in previous examples was to display the variables coming in from the input data set, now we won't want to have to maintain that list either.

```
data discount;
     set CurrentPrice;
     array Products {*} _Numeric_;
     do j = 1 to dim(Products);
            Products{j}=Products{j}*(1-.1);
```

```
     end;
run;
```

Now our program is ready for any number of Product price variables, the new Work.Discount data set will have the modified, discounted prices. If you need the NewPrice variables, simply define the second array and modify the assignment statement.

CHARACTER ARRAYS

An array can also refer to a group of character variables that need to be treated in the same way. With each of our Product variables that contain the price, we will assume there is also a Product code variable (Prod_Code1-Prod_Code4) that contains a string of information on who the manufacturer is, where the product was manufactured, weight and dimensions. For Reporting and Analysis, our business requirements are now to expand the information in that column to include some information from the Location variable. This location variable contains the full mailing address as to where the product is being sold, keeping in mind that the prices for these retail products vary with location as might sales tax. For each observation there is a location variable, and for each Product, a price variable and a product code variable. In this example, we will define a character array to concatenate the two-character State or Province code from the Location variable to the end of the Product code variable. To think of this the long way first may help, then go back and update the program to utilize an array. The format of the location variable looks like this for example, "123 First Street,Toronto,ON,Canada,A1B 2C3" so we are looking for the third field using the comma as a delimiter. To extract this string (also considered a 'word' within the value), the SCAN function can be used and has three arguments. The arguments are in order and specify the 1) character string or variable to extract from 2) which word by number to extract starting from the left and 3) what character will be the delimiter separating 'words' in the value. The Prod_Code variable has a series of codes separated by underscores, so for our example we will want to also concatenate an underscore before appending the State/Province code (using two exclamation marks !! as the concatenation operator).

```
data CodesUpdate;
     set CurrentPrice;
     STPR_code=Scan(Location,3,",");
     NewProd_Code1=Prod_Code1!!'_'!!STPR_code;
     NewProd_Code2=Prod_Code2!!'_'!!STPR_code;
     NewProd_Code3=Prod_Code3!!'_'!!STPR_code;
     NewProd_Code4=Prod_Code4!!'_'!!STPR_code;
run;
```

Again, the more product variables we have the longer the code becomes and also, as we add or remove products this code needs to be constantly maintained. Having seen the pattern, now we can recode using an array:

```
data CodesUpdate;
     set CurrentPrice;
     array Prod_Code {4};
     array New {4} $ 50 NewProd_Code1 -
     NewProd_code4;
     do j = 1 to 4;
     New{j}=Prod_Code{j}!!'_'!!Scan(location,
     3, ',');
     end;
run;
```

On the Prod_Code ARRAY statement, notice that even though we have not specified a $, this is a character array since the variables it will reference (Prod_Code1 – Prod_Code4) are defined as character and already in the PDV when the ARRAY statement is compiled. To save another assignment statement we have moved the scan function that extracts the State or Province code from the location onto the concatenation expression. Since we were defining the NewProd_Code array to reference a group of new product code variables, the ARRAY statement needs to include the $ to indicate the variables will be character and also specify a length (50 in this example). If the length is not specified on the ARRAY statement for a character array where the variables do not yet exist in the PDV at compile time, they will be given a default length of 8 bytes which of course in our example would lead to truncation.

There is also a corresponding keyword _CHARACTER_ that can be used in the same way the _NUMERIC_ was, except of course it will align all the character variables in the PDV at the time when the ARRAY statement is compiled to the elements of that array. To build an array using this technique to reference all of the Prod_Code character variables, keep in mind you would have to drop the Location variable if it was not needed or, only process the desired elements of the array starting at the 2nd element (assuming Location was the first character variable in the array).

```
data CodesUpdate;
     set CurrentPrice;

     array Prod_Code {*} _character_;
     array New {4} $ 50 NewProd_Code1 -
     NewProd_code4;

     do j = 2 to dim(Prod_Code);
     New{j-1}=
     Prod_Code{j}!!'_'!!Scan(location, 3,
     ',');
     end;
run;
```

SETTING INITIAL VALUES

Each individual array can only be defined as character or numeric but within a data step it can be quite powerful to define several separate arrays with some being numeric while others are character. Arrays can also help to shorten a program that has a large number of conditions to test in order to assign a value. In our retail example, the taxes that are charged or applicable varies between states and provinces and our requirements now call for a variable that multiplies the price by the tax to calculate the total price. For each observation or location, our program would need to test over 60 conditions to determine the state or province and then take the price and multiply it by the appropriate tax multiplier. This could be quite lengthy, for this example, we will only show a few states and rather than specify accurate tax rates, our program will refer to them as simply multipliers and will be a number less than 10%.

```
data Multiplier;
     set CurrentPrice;
     STPR_code=Scan(Location,3,",");
     if STPR_code='BC' then
     do;
     total_1=prod1 * 1.07;
     total_2=prod2 * 1.07;
     total_3=prod3 * 1.07;
     total_4=prod4 * 1.07;
     end;
     else
     if  STPR_code='NJ' then
     do;
     total_1=prod1 * 1.08;
     total_2=prod2 * 1.08;
     total_3=prod3 * 1.08;
     total_4=prod4 * 1.08;
```

```
      end;
      else
      if  STPR_code='FL' then
      do;
      total_1=prod1 * 1.04;
      total_2=prod2 * 1.04;
      total_3=prod3 * 1.04;
      total_4=prod4 * 1.04;
      end;
      if  STPR_code='ON' then
      do;
      total_1=prod1 * 1.06;
      total_2=prod2 * 1.06;
      total_3=prod3 * 1.06;
      total_4=prod4 * 1.06;
      end;


        /* and on and on for all
           the states and provinces
        */
      run;
```

As store locations expand across the US and Canada, this program becomes very lengthy and contains predominantly repetitive code. But how can arrays help us to reduce this program when we need to test each observation to see what the state or province code is? In order to accomplish this task we will use the ability to define initial values in an array that is defined in our data step. We will use these new arrays slightly differently in that we won't actually be using the arrays to refer to variables, rather we will use them as a "virtual" table that will hold in one array the values of the states/provinces and in another their corresponding multiplier rate. Iterative and conditional processing will allow us to assign the proper multiplier rate to be applied to all the product price variables creating a new total variable.
The initial values are included after the Elements are specified on the ARRAY statement and are enclosed in parentheses. These values will be automatically retained.

```
data Multiplier;
     set CurrentPrice;
     array Prod {4};
     array total_ {4};

     array ST_PR{4}$ ('BC', 'NJ', 'FL', 'ON');
     array mult{4} (.07,.08,.04, .05) ;

     do j= 1 to 4;
     if Scan(Location,3,",")=ST_PR{j} then
     multiplier=mult{j}+1; *gives us 1.07 etc;
     end;

     do k= 1 to 4;
     total_{k}=prod{k}*multiplier;
     end;
run;
```

With 3 numeric arrays, 1 character array, iterative and conditional processing, this program is considerably easier and shorter to code. It could be argued that the long version might actually run faster due to the long list of conditions being linked with an ELSE statement. With the ELSE statement, as soon as SAS encounters a true condition, the remaining condition tests linked with an ELSE are not tested. So, programmer efficiency or runtime efficiency, which would it be? Why not both?

If we use a slight variation on the do loop we can achieve the same efficiency gain without the else statements. It might also be more efficient to include the separate assignment statement that determines the state/province code rather than rescanning this for every evaluation of the IF condition.

```
data Multiplier;
     set CurrentPrice;
     array Prod {4};
     array total_ {4};

     array ST_PR{4}$ ('BC', 'NJ', 'FL', 'ON');
     array mult{4} (.07, .08, .04, .05) ;

     STPR_code=Scan(Location,3,",");
     stop=0;

     do j= 1 to 4 until (stop=1);
     if STPR_code =ST_PR{j} then
      do;
      multiplier=mult{j}+1;
      stop=1;
      end;
     end;

     do k= 1 to 4;
     total_{k}=prod{k}*multiplier;
     end;
run;
```

In this program, the DO statement includes an UNTIL condition that allows iterating through the loop until a condition is met and this condition is tested at the bottom of the DO loop. For each observation read in from the input data set, stop is set to zero and the array subscript variable j is set back to 1 upon entering the loop. If the value of the state or province code is equal to the value in the current array reference then the corresponding multiplier rate is assigned to the variable for subsequent calculation. The variable STOP is set to 1 which will terminate this loop saving testing the remaining comparisons. On the DO statement, the condition that will stop the loop could be either the UNTIL condition or the counter variable J, having both conditions guards against the case where the value extracted in STPR_code is not in the ST_PR array and STOP would not get set to 1. If this occurred due to invalid data or a new state that was not in the data since the last update to this program, the subscript variable j would not exceed the dimension of the array and therefore not result in an ERROR in the program.

At this point if we run a Proc Contents on the WORK.MULTIPLIER data set we would find many variables that we don't necessarily need. Remember that at compile time an ARRAY statement either aligns an array reference to variables that already exist in the PDV or it creates new variables. In our example here, the ST_PR and Mult arrays are not needed as variables in the output data set and would take up considerable room being added to every observation. The data set would also contain the index variables j and k as well as the stop variable we are using for the efficiency gain. One approach would be to use a DROP= data set option on the DATA statement or a DROP statement within the data step. This is what we will do for the index variables and the stop variable. For the arrays however, if we code them on the option or the DROP statement, it will be potentially lengthy and as our list of states and provinces grow or shrink, it would become one more part of our program to maintain.

To help in this situation, there is a keyword that can be included on the ARRAY statement in place of element names. The keyword is _TEMPORARY_ and what it signals to SAS is that is does not need to create actual variables in the PDV for this array and that the elements of the array will be held in memory but not output as variables to the data set. In a sense we are using these two arrays then as lookup tables to help us with our processing. It is important to note that using _temporary_ also

provides efficiency gains in processing, taking less memory to store and in some situations, less CPU to process.

```
data Multiplier(drop=j k stop);
     set CurrentPrice;
     array Prod {4};
     array total_ {4};

     array ST_PR{4}$ _temporary_ ('BC', 'NJ',
     'FL', 'ON');
     array mult{4} _temporary_ (.07, .08, .04,
     .05) ;

     STPR_code=Scan(Location,3,",");
     stop=0;

     do j= 1 to 4 until (stop=1);
     if STPR_code =ST_PR{j} then
      do;
      multiplier=mult{j}+1;
      stop=1;
      end;
     end;

     do k= 1 to 4;
     total_{k}=prod{k}*multiplier;
     end;
run;
```

To take it one final step further by incorporating a previous technique, we could use the asterisk to specify the dimension of the arrays and the DIM function to set the stop value for the incremental do loop as follows:

```
data Multiplier(drop=j k stop);
     set CurrentPrice;
     array Prod {*} _numeric_;
     array total_ {4};

     array ST_PR{4}$ _temporary_ ('BC', 'NJ',
     'FL', 'ON');
     array mult{4} _temporary_ (.07, .08, .04,
     .05) ;

     STPR_code=Scan(Location,3,",");
     stop=0;

     do j= 1 to 4 until (stop=1);
     if STPR_code =ST_PR{j} then
      do;
      multiplier=mult{j}+1;
     stop=1;
      end;
     end;

     do k= 1 to dim(Prod);
     total_{k}=prod{k}*multiplier;
     end;
run;
```

TRANSPOSING DATA

Depending on the type of analysis that needs to be done, the data may need more work than just subsetting, transformation or creation of new variables. Often in reporting or Data Mining, there is a need to merge many files together or take existing data and rotate or transpose it so that information is arranged in a different structure. There may be a need to take information that was spread down many observations in one column and rotate it so it becomes aligned across one row. Of course, transposing data can also work the other direction where you take values in the rows and rotate them to be in one column. In SAS, there is a very powerful and flexible procedure for this: PROC TRANSPOSE. Using arrays to rotate data can provide efficiency gains when you need to do multiple transposes, but we will focus on using arrays to see a simple example of rotating data.

Working again with our retail data, another business requirement has arisen where we need to look at our data grouped by product. With the current structure of the data if we wanted to produce a graph to show the average price of each product across all locations a simple PROC GCHART wouldn't show us this. In this case we need to change the structure of the data from one long observation with all the product prices for a location to an observation for each location, the product name and its price. The change in the structure of the data would be something like the following:
Original structure …

LOCATION	PROD1	PROD2	PROD3	PROD4
123 My St,City,NJ,USA	319.43	89.99	149.55	17.99

New structure …

LOCATION	Product	Price
123 My St,City,NJ,USA	PROD1	319.43
123 My St,City,NJ,USA	PROD2	89.99
123 My St,City,NJ,USA	PROD3	149.55
123 My St,City,NJ,USA	PROD4	17.99

Now, with a PROC GCHART or similar procedure, we could chart the Product variable and request the sum or mean statistic on the price variable and see a bar chart for the average price of all PROD1 (Television) prices across the locations. To assign a Product name to the generic Prod variables and rearranging the data without an array can be done but would involve a long repetitive program.

```
data longRotate(keep=Location
                     Product
                     Price);
     set CurrentPrice;
     Length Product $ 12;

     Product='Television';
     Price= Prod1;
     output;

     Product='Radio';
     Price=Prod2;
     output;

     Product='MicroWave';
     Price=Prod3;
     output;

     Product='Toaster';
     Price=Prod4;
     output;
run;
```

To write this with an array and some of the techniques we've seen so far will greatly reduce the redundant code, still provide a descriptive value for the Products and enhance the ease of maintenance for future additions of products.

```
data ShortRotate(keep=Location
                      Product
                      Price);
     Set CurrentPrice;
     Array Prod {*} _numeric_;
     Array Product_Names {4} $ 12
```

```
                    ('Television',
                     'Radio'
                     'Microwave'
                     'Toaster');

      do j= 1 to dim(Prod);
       Product=Product_Names{j};
       Price=Prod{j};
       Output;
      end;
run;
```

As the number of products grows we will need to add their descriptive name to the initial values of the Product_Names array or, if this mapping of Prod1 to Television existed in another data source, it could be merged in to further automate the process.

CONCLUSION

Whenever there are a group of variables to be processed in the data step, it might be well worth considering using arrays to help accomplish the business objective. In some cases, arrays will help to simply reduce redundant code and this provides the opportunity for SAS programmers to either revise existing programs and applications or to start using arrays in future development.

Planning your code is important in any project. In planning a data step, you may start by writing out a skeleton of the code needed using a long form approach, simply following the business rules. If you look at some of the examples in this paper, they show the long approach then the shorter approach with arrays. This may in fact be the steps you choose to follow in planning your code as it will allow you to define the business process and rules then look for ways to shorten the coding process via arrays. It won't always be immediately evident at first how you could use and benefit from arrays quite often until you see the code starting to take up a full page and the repetitive call to different variables to be processed in the same way.

Shorter programs don't always run faster (sometimes they do!) but they can be easier to maintain and arrays do have many nice features that allow programs to approach the "maintenance free" status. The objective of this paper was to introduce and get you "…off and running with arrays in SAS" by understanding the benefit and how to implement them in your programs. There are more advanced features of arrays such as multidimensional arrays and also the SAS Macro Language is a powerful addition allowing us to build data driven code.

ACKNOWLEDGMENTS

Special thanks to my colleague and mentor William Fehlner Phd. for help in reviewing this paper. Also of special note are two papers from previous SUGI's written by Marge Scerbo (Paper 55-25 and 52-26).

CONTACT INFORMATION

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

Stephen Keelan
SAS Institute (Canada) Inc.
181 Bay Street, Suite 2220
Toronto ON M5J 2T3
Work Phone: (416) 307-4592
Fax: (416) 363-5399
Email: Stephen.Keelan@sas.com
Web: www.sas.com/training