Array Programming Basics

John J. Cohen, AstraZeneca LP, Wilmington, DE

Abstract
Using arrays offers a wonderful extension to your SAS® programming toolkit. Whenever iterative processing is called for they can make programming easier and programs easier to maintain. You will need to learn some new syntax, but we will explain several of the key components such as indexing and subscripts, temporary and multi-dimensional arrays, determining array dimension, and a few special tricks.

Introduction
Arrays are a programming convenience, offering us an opportunity to organize a set of variables as a collective and to then refer to these as that single collective in subsequent programming. Program execution may not be more efficient (although we will be talking about temporary arrays). The activity of writing and maintaining programs, however, can be helped immeasurably. We will also discuss certain tasks made vastly easier by using arrays. We will offer here the basics, and when we are done you will be well equipped to explore using arrays in your programming. At the Conference we will provide certain advanced examples.

Why Arrays
In SAS arrays are used to group variables together. This allows us to perform the same action across the entire grouping at once rather than having to specify our process for each variable individually. For instance, let's say we have a data set consisting of the latest 52 weeks if sales data for each of 500 territories. We want to check each variable for a missing value and if we find one, turn it into a zero. Most likely we would try something like the following in Figure 1.

Figure 1 – Why Arrays – The Old Fashioned Way

```
Data Sales;
   set Sales;
   if week01 = . then week01 = 0;
   if week02 = . then week02 = 0;
   if week03 = . then week03 = 0;
   if week04 = . then week04 = 0;
   if week05 = . then week05 = 0;
   if week06 = . then week06 = 0;
   if week07 = . then week07 = 0;
   if week08 = . then week08 = 0;
   if week09 = . then week09 = 0;
   if week10 = . then week10 = 0;
   if week11 = . then week11 = 0;
   if week12 = . then week12 = 0;
   if week13 = . then week13 = 0;
   if week14 = . then week14 = 0;
    .   .    . .   .    .    . .
    .   .    . .   .    .    . .
    .   .    . .   .    .    . .
   if week50 = . then week50 = 0;
   if week51 = . then week51 = 0;
   if week52 = . then week52 = 0;
run;
```

There is nothing wrong with this per se. Our program will quickly and efficiently perform the intended task. However, putting this simple step together proved tedious and surprisingly error-prone. (Do I have exactly fifty-two statements? Did I mistype any of the variable names, or in editing copied lines, get spacing incorrect?) Even minor changes and corrections are problematic. (Do I need to change the variable names from Week01, etc. to WK01, etc.?) Viewing program flow is much harder when the program step will not fit on a single page or screen. And here we are only deal with fifty-two variables. Imagine if the data contained, say, 365 variables (one for each day of the year) or 10,000. We would quickly seek a neater solution.

What Arrays Are

Arrays are SAS statements containing the following elements:

Key Word	Array Name	Number of Elements	List of Variables
Array	Weeks	{52}	Week01 – Week52;

The key word ***Array*** signals to SAS that we are defining an array. Array names fit most typical SAS variable naming rules except that you may not use a variable name being used in that same array as an array name. The number of elements corresponds to the number of variables in our list. Finally, we list the variables that we wish to group together. We might address our Sales data set as follows:

```
Array Weeks {52} Week01 – Week52;
```

We could have as correctly coded the variable list as shown below, but this would tend to defeat the purpose of having easy to understand programs.

```
Array Weeks {52} Week01 Week02 Week03 Week04 . . . Week50 Week51 Week52;
```

The task from Figure 1 might look as shown in Figure 2 using an array.

Figure 2 – Why Arrays – A Simple Array

```
Data Sales;
   set Sales;
   Array Weeks {52} Week01 – Week52;
   do i = 1 to 52;
      if Weeks{i} = . then Weeks{i} = 0;
   end;
run;
```

We declare our array Weeks with fifty-two elements. Over each of the individual elements we wish to perform the same task, checking for a missing value and changing any found to a zero. We can refer to the first element of our array as Weeks{1} (or Week01), the second as Weeks{2} (or Week02), and so on through Weeks{52} (or Week52). To address each of the elements we use a do/end loop and employ an index variable (by convention often called “i”). (For each value of the index variable i from 1 to 52 we perform our check.) Thus, with vastly simpler code to write and read (and seven program lines vs. fifty-five lines) we achieve the same task.

Additional Details - Variable List Formats and Naming Conventions
Variable lists need not be numeric-suffixed, as is the example above -- this is merely a helpful programming convention. We might refer to variables for the days of the week in an array such as

```
Array Week {7} Monday Tuesday Wednesday Thursday Friday Saturday Sunday;
```

Our variable list can be empty (actually, implied) as in the following example.

```
Array month {7};
```

SAS will create for us an implied list of month1 through month7. If any of these variables did not exist already they will be created for us and retained.

Our variable list could be supplied to us as the value of a macro variable such as

```
Array cities {*} &variable_list;
```

where the value of macro variable &variable list might be “Boston Chicago Miami” one time and “Atlanta Houston Philadelphia Reno Toronto” the next. Or the number of elements might be set by a macro variable in the following manner where the patient count in the incoming data set is passed to the array step as the observation count.

```
Array patients {*} patient_id01 – patient_id&count;
```

We can refer to all numeric variables with the keyword _numeric_ as in the following:

```
Array numerics {*} _numeric_;
```

Similarly, all character variables can be referred to as _character_

```
Array characters {*} _character_;
```

Array Indexes
Note that in several examples above we did NOT specify the number of elements in our variable list. Instead we asked SAS to count this for us by using an asterisk (“*”) inside the braces (the “{“ and “}”). This is helpful when we may not be sure of the number of variables in our list and the values of the index variable have no particular meaning. (This would be the case in the examples above using macro variables to help describe the variable list or when using the _numeric_ or _character_ key words.)

In another set of circumstances, more explicit control of the values of the index variable is required. SAS, unlike many other programming languages, starts array index variables with the value of “1”. (Many other languages start with “0”.) Let’s say our annualized sales data consisted of nineteen yearly totals starting in 1987. Our variables might be named Year1987, Year1988, etc. and our array might look like this:

```
Array Years {1987:2005} Year1987 – Year2005;
```

Our programming needs may be such that we prefer to refer to Years{1987} through Years{2005} rather than to Years{1} through Years{19}. By specifying the starting point of 1987 and ending point of 2005 within the braces

```
{1987:2005}
```

we may have access to better program design – or at least something more self-documenting.

Final Thoughts on the Array Statement

Arrays are a temporary construct, present only for the duration of the Data step in which they reside. If any variables are created in the use of the array, however, these become permanent variables. Braces are no longer required, an artifact of efforts to maintain compatibility across multiple operating systems in earlier SAS releases. (Brackets – [] and parentheses () are now acceptable.) We should also note that the array index variable must be numeric and that it becomes another permanent variable in your output data set unless you drop it.

Figure 3 – Dropping the Index Variable

```
Data Sales;
   set Sales;
   Array Weeks {52} Week01 – Week52;
   do i = 1 to 52;
      if Weeks{i} = . then Weeks{i} = 0;
   end;
   drop i;   /** we will not need i after end of data step **/
run;
```

Temporary Arrays

Temporary arrays, indicated by the option _temporary_, are arrays which has the variables available only during the data step in which the array is active. If only needed during the duration of the data step, then declaring these as temporary precludes the need to drop these variables. Further, as these will reside in memory (rather than being read from/written to disk), certain processing efficiencies are possible, especially when working with very large data sets. (See Dorfman for an example in Bit Mapping.) Note that these all must be temporary variables – none existed already in the incoming data set – and the number of elements must be specified. These might be used for accumulating intermediate values. Notice also that we are able to initialize the values of the variables in our temporary array as we do immediately below. (See Wright for a number of lovely examples of initializing variable values at array definition time.)

Figure 4 – Temporary Arrays

```
Data Adjusted_monthly_sales;
   Set monthly_sales;
   Array sales {6} monthly_sales1 – monthly_sales6;
   Array adjust {6} _temporary_ adj_1 – adj_6 (.7 .2 .3 .4 .2 .3);
   do i = 1 to 6;
      sales{i} = sales{i} * adjust{i};
   end;
   drop i;
run;
```

In this Figure 4 we define two arrays, one for the incoming monthly sales figures and our temporary array initialized to contain the monthly adjustment amounts. After adjusting the incoming sales figures, the temporary array variables will automatically be dropped. We explicitly drop our index variable I, leaving a new data set with adjusted monthly sales figures.

Multi-dimensional Arrays

Often more advanced uses of arrays are about the structure of your data. Rather than referring to your data as a single string of variables, a more intuitive approach may be to think of your data in terms of, say, rows and columns, or two dimensions. (We could go many more.) If we have three years of monthly sales data, thinking of your data as three years (rows) of twelve months (columns) may be a more meaningful way to proceed than as 36 months of data. In Figure 5 we have represented twelve months (in columns) of sales data for each of three years (in rows). Note that the data are missing in month12 for the year 2005. (These data are captured, of course, before year's end.)

Figure 5 – Multi-dimensional Arrays – the Data Structure

Year	Month1	Month2	Month3	Month4	Month5	. . .	Month12
2003	274	345	123	234	451		347
2004	235	178	166	278	287		116
2005	198	456	321	99	231		.

Our array statement will account for the two dimensions by indicating the number of elements in first the rows, then columns. However, the data are actually contained in variables Y2003_M1 through Y2003_M12, Y2004_M1 through Y2004_M12, and Y2005_M1 through Y2005_M12. Our data from Figure 5 might be defined as

```
Array months {3,12} Y2003_M1 - Y2003_M12 Y2004_M1 - Y2004_M12
                    Y2005_M1 - Y2005_M12;
```

This will allow us to address our data in this more intuitive format. We will use two do/end loops to traverse the two dimensions as shown in Figure 6.

Figure 6 – Multi-dimensional Arrays – the Program

```
Data Sales;
   Set sales;
   Array months {3,13} Y2003_M1 - Y2003_M12 Y2003_total
                       Y2004_M1 - Y2004_M12 Y2004_total
                       Y2005_M1 - Y2005_M12 Y2005_total;
   do i = 1 to 3;
      months{i,13} = 0;   /** initialize yearly totals to zero **/
      do j = 1 to 12;
         if months{i,j} = . then months{i,j} = 0;
         months {i,13}= months {i,13} + months {i,j};
      end;
   end;
   drop i j;
run;
```

Briefly, we define our array *months* in two dimensions, three years (the rows) and twelve plus one months (for a total of thirteen columns). The additional variable in each row will capture the annual totals. We use two Do/end loops and two index variables (i and j). At the beginning of each year/row (from row 1 to row 3) we initialize the annual total variable for that year (or row

number i, column 13). Next for each value of j (from 1 to 12) we perform our missing values check, then accumulate the individual months.

The program in Figure 6 takes our 36 months of data and processes the variables in blocks of 12. We could specify the calculation of an annual total once for each year (after turning any missing values into zeros) but this approach will allow us handle 3 years, 23, or 103 we equal ease. Note that the resulting data set will still contain our original 36 months of data plus an annual total for each of the three years. We will discuss below a method for restructuring the data using arrays.

Array Functions

Three special array functions are available to us, all addressing the identification of the number of elements in our array. The Dim function returns the dimension of an array and has the following syntax:

dimension of array = dim(*array name*);

This is useful in circumstances when we have used the asterisk to indicate the number of elements in an array.

Figure 7 – the DIM Function

```
Data Sales;
   Set sales;
   Array weeks {*} weeks1 – weeks&count;
   do i = 1 to dim(weeks);
      /*** additional program statements ***/
   end;
run;
```

In the case of a multi-dimensional array, dim2(*array name*) will return the dimension of the second dimension, dim3(*array name*) the third, and so on.

Other functions are Hbound and Lbound, returning, respectively, the upper and lower bounds of a dimension. Where the array index does not necessarily start with a value of one or the upper index value is not identical to the number of elements in the array, these can be invaluable. As from our example above, the index values for the array Years would be processed as follows:

Array Years {1987:2005} Year1987 – Year2005;

Function Call	**Value Returned**
Dim(years)	19
Lbound(years)	1987
Hbound(years)	2005

Error Debugging

Arrays introduce another set of error messages and a whole host of program validation issues which, sadly, may NOT always trigged error messages to warn us. As in any programming process, starting and testing incrementally and borrowing working examples are helpful steps to being successful. Another useful technique is to test with one, or perhaps just a handful of observations, printing off the values of variables at different points in your data step. Printing values to the log can be accomplished with a “Put _ALL_; “ statement located strategically within

your data step. If there are too many variables to easily decipher your process, a more limited set of Put statements might look like the following:

```
Data sales;
      /** additional statements **/
  Put “at line 12 ”  _N_= i= month1= month2= total= ;
      /** additional statements **/
  Put “at line 18 ”  _N_= i= month1= month2= total= ;
      /** additional statements **/
Run;
```

Depending on the logic of your data step, these statements will print to your log something resembling:

```
At line 12 _N_=1 i=1 month1=17 month2=24 total=41
At line 18 _N_=1 i=2 month1=17 month2=24 total=157
```

A common class of problems in using arrays center around improper accumulation across the running of the data step. This can happen because a variable in question is not properly initialized (and thus you are trying to add missing values to non-missing values). Alternatively, you may not be re-setting your accumulator variable(s) to zero at the culmination of a particular block of data. These tend to not trigger error messages and instead require extensive validation of your program. In the former case, a clear warning sign would be the following note in your log:

NOTE: Missing values were generated as a result of performing an operation on missing values.

A second set of issues common to use of arrays regards the array subscript. These can be quite puzzling to correct but are more likely to trigger error messages and thus are more obvious when they arise. Rules of thumb include avoiding using the same index variable twice within the same data step, confirming that your index variable is not an already-existing variable which already has values, and again, build your program incrementally. An error message such as in Figure 8

Figure 8 – Array Subscript out of Range

```
16  data test;
17     set sales(keep=month1-month5);
18     array months {*} month1-month5;
19     do i = 1 to 6;
20        months{i} =  months{i}*10;
21     end;
22   run;

ERROR: Array subscript out of range at line 20 column 28.
month1=1880 month2=6940 month3=7990 month4=5260 month5=420 i=6
_ERROR_=1 _N_=1
```

offers immediate clues as to where we look next. SAS points us to the exact line where the error occurred (line 20), which observation triggered the error (the very first: _N_ = 1) and what the value of the subscript (the value of the index variable i) at the point that the error was triggered. In this instance the value of i is 6. As we have defined an array with only five variables, doing i = 1 to 6 will push the subscript outside of the defined bounds for our array. In this simple example,

changing the Do/end loop to do I = 1 to 5 will allow this data step to execute without errors. Of course, you may have had a valid reason for wanting the do loop to process (or *iterate*) six times, so revisiting your logic may be required to get the desired results.

Applications/Examples

We will offer here three examples of programs using arrays to perform tasks that might be harder without. The first queries our data set for observations with every value missing. If so, then the observation is deleted.

Example 1 – All Missing

Figure 9 – All Missing

```
Data sales;
   Set sales;
   All_missing = 1;  /** initialize to one (“yes”) **/
   Array months {12} month1 – month12;
   do i = 1 to 12;
      if months{i} NE . then all_missing = 0;
   end;
   if all_missing then delete;
run;
```

The program logic works “in reverse.” A logical variable (values of “0” or “1” only), all_missing, is initialized to “1”. The data step iterates through each of twelve months to check for missing values. Any single instance of a non-missing value will cause the all_missing flag to be switched to “0” (or “no”). Our final query, after the completion of all twelve iterations, deletes any observation in which we did NOT encounter a non-missing value and thus did not flip the switch (i.e., all values for that observation were missing.) Once understood, this is certainly more elegant than something like:

If month1 = . and month2 = . and month3 = . etc.

Example 2 – Transposing Your Data Set

Figures 10, 11, and 12 offer an example of restructuring a data set, transposing some of the columns to rows. (We could as easily use this same approach to go in the other direction, transposing rows to columns.) Proc Transpose is a common alternative approach but has limitations in certain instances (in addition to the warning that “it may produce unexpected results). Of particular note is that Proc Transpose can only transpose your data – often completely sufficient. If however, you will need to perform considerable processing on the data in addition to performing a transpose, using a single data step to accomplish the entire task may be preferable.

A typical programming problem is our incoming data are transactional in nature, perhaps one or more observations per customer per month, for instance. We would like to have a single observation for each customer reflecting all twelve months. To do so, a common approach, is to sort the data set by customer id, then *set by* customer id in the data step. This allows us to use first. and last. processing. (These are logical variables automatically created by SAS and available for the duration of the data step.)

Figure 10 – Transposing Your Data Set – the Incoming Data

Customer Id	Sales_Amount	Date
1	$27.50	12/02/05
1	$27.50	12/03/05
2	$47.50	02/05/05
3	$14.25	12/02/05
3	$14.25	14/05/05
3	$37.14	27/06/05
4	$45.00	06/01/05
6	$17.27	30/03/05

We have an incoming data set consisting of customer id, dollar sales amount, and the sale date. We will sort the data by customer id, create an array of outgoing variables, and will use a do/end loop to transpose the data. We expect to end with fewer observations – but more variables – than when we started.

Figure 11 – Transposing Your Data Set – the Program

```
Proc sort data=sales; by customer_id; run;

Data annualized_sales;
   Set sales;
   by customer_id;           /** allows first. and last. processing **/
   Retain month1 – month12 0;
   Array months {12} month1 – month12;
   if sales_amount = . then delete;
   index = month(date);           /** extract month from date value **/
   months{index} = months{index} + sales_amount;
   if last.customer_id then do;   /** last observation for customer **/
      YTD_total = sum(of month1 – month12);  /** year-to-date total **/
      output;               /** output one observation per customer **/
      do i = 1 to 12;
         months{i} = 0;         /** reset to zero for next customer **/
      end;
   end;
   drop i sales date;
run;
```

We *retain* our array of monthly buckets so that we can accumulate the values across multiple incoming observations. We do not use a do/end loop immediately, instead opting for *implicit* subscripting. (This is NOT the same definition of *implicit arrays* from back in the olden days before PCs were invented.) We extract the month from the date and then move our subscript pointer as indicated by the month values in the incoming data set. (We initialize all months to zero at the outset as there is not a guarantee that each customer has made a purchase in every month – we will have zeros placed in those buckets rather than missing values.) As we *set by* customer id, we can then ask SAS to check for last. or the last observation for each customer. When the last observation for a customer is encountered, we calculate a year-to-date total and only then output a single observation for the customer. We then reset the 12 monthly buckets to zero. We drop sales and date – these should be captured in month1 – month12. (In testing, of course, you will likely want to keep these for initial validation.)

Figure 12 – Transposing Your Data Set – the Resulting Data

Customer Id	Month1	Month2	Month3	Month4	. . .	Month12	YTD_total
1	$0.00	$27.50	$27.50	$0.00		$0.00	$55.00
2	$0.00	$0.00	$0.00	$0.00		$0.00	$47.50
3	$0.00	$14.25	$0.00	$0.00		$0.00	$65.64
4	$45.00	$0.00	$0.00	$0.00		$0.00	$45.00
6	$0.00	$0.00	$17.27	$0.00		$0.00	$17.27

In this simple example we go from eight observations and three variables to five observations and fourteen variables. In a more typical setting, our data would not be this sparse. (Note, however, that the use of implicit subscripting means that for sparse data, we will fill the respective buckets very efficiently.) While combining a Proc transpose step with another step of some kind could likely offer similar results, from the standpoint of logical flow it may be easier to combine several steps into one data step, particularly if the goal is to output several levels of consolidation, some of it conditional upon incoming dynamic values.

Example 3 – Non-Numeric Index Variable

As indicated above, the index variable must be numeric. Converting character to numeric is the sole additional step required to make your program work. In Figure 13 we take a data set containing postal abbreviation codes (e.g., AK, AR, CA, NE, MA, WY, PA, etc.). We wish to process some of the incoming data and output the results into variables names consisting of those same state abbreviations. By using the stfips function supplied by SAS, we accomplish our task, converting the character variables into numbers corresponding to our index. (Of course, we must be sure to list the state abbreviations on our array statement in exactly the US Postal Fips Code order!)

Figure 13 – Character Index Variable

```
Data States;
   Set states;
   Array states {*} AK AR AZ CA  . . .  WY;
   index = stfips(state_abbreviation);  /* convert NC to 37, etc. */
   states{index} = /** additional statements **/
   output;
run;
```

Similarly, we could extract the month value from a character date field (of the format ddmmyy such as “14/02/05”) and convert to numeric as follows:

```
index = substring(character_date,4,2) * 1;
```

Conclusion

We have gone through the basics of array programming and seen that arrays provide a valuable addition to our SAS programming tool kit. In some instances arrays provide us merely with a programming shorthand. These may offer no processing improvements, but easier code to write and maintain (in an era where programmer time is nearly priceless). In other instances arrays allow us to take on problems that would be more difficult without using them. Finally, especially temporary arrays can improve processing efficiencies in handling large data sets. The productive use of arrays only requires practice, interesting problems, and your imagination.

References and Resources

Paul M. Dorfman (Animated and Presented By Russell Lavery), "An Animated Guide: Speed Merges with Key indexing, Bitmapping and Hashing," NESUG 2002, September 29 – October 2, 2002, Buffalo, NY.

SAS Institute, Inc., **SAS® Guide to Macro Processing, Version 6, Second Edition**, Cary, NC, SAS Institute, Inc., 1990.

SAS Institute Inc., **SAS® Language: Reference, Version 6, First Edition**, Cary, NC: SAS Institute Inc., 1990.

Wendi L. Wright, "Loop-Do-Loop Around Arrays," NESUG 2005, September 11-14, 2005, Portland, ME.

In addition, SAS provides a number of sample programs (search on "array") at its user/technical support site:

http://support.sas.com/ctx/samples/index.jsp

Trademarks

Author

John J. Cohen
Senior National Segment Analyst
AstraZeneca Pharmaceuticals
1800 Concord Pike, RL6-415C
Wilmington, DE 19850-5437
(302) 886-7083
john.cohen@astrazeneca.com