Paper 118-2010

A New Look At An Old Friend: Getting Reacquainted With Arrays

Joe Novotny, dunnhumby USA, Cincinnati, OH
Paul Kollner, dunnhumby USA, Cincinnati, OH

ABSTRACT
Though SAS® DATA Step Arrays have been around for a long time, they are one of the more elusive constructs in the SAS System. Even many experienced SAS programmers are less than proficient with these powerful SAS programming tools. This paper springs out of this very scenario and attempts to remedy the situation for one of the authors. We start at the beginning, defining what Arrays are. We discuss Array characteristics and uses at an introductory level, and then proceed on to more advanced topics. The paper is meant to serve as a solid introduction for those uninitiated in Array processing as well as a review and deeper dive into some more intricate topics for more advanced SAS programmers. We review and provide examples of Array definition and processing for the following: 1) one and two dimensional Arrays, 2) explicit vs. implicit Arrays, 3) _temporary_ Arrays, 4) iterative processing with do-loops and indexing and 5) comparison of Array processing with SAS Data Step hash objects. Applying the concepts introduced in this paper will improve your code's efficiency as well as make it more flexible and maintainable for future use.

BACKGROUND AND ASSUMPTIONS
Becoming familiar with SAS Array processing adds a powerful tool to any SAS Programmer's toolbox. Array processing is a component of the BASE SAS System and all examples presented in this paper have been developed by the authors using BASE SAS software v9.1.3 (SP 4). The intent of this paper is to demonstrate *practical* examples of Arrays that the authors have found useful. Please note that all data and examples have been recoded to ensure anonymity and preservation of intellectual property rights. Introductory material will be presented at the front of the paper and will be built upon so that as we dig further into the topic, more advanced applications of Arrays will become clearer.   While an introductory understanding of BASE SAS is assumed, all topics integral to understanding the Array processing examples will be introduced and explained within the paper. In addition, we hope you find some ways of using SAS you may not have considered before - coding snippets to add to your SAS toolbox to make you a better programmer (and more importantly, impress your colleagues!). The techniques described in the paper were developed through experience, consultation with colleagues and referral to the SAS documentation. We hope new users find the appropriate level of information to get started and more experienced users find a new perspective on the topic of Arrays. Happy coding!

WHAT IS AN ARRAY?
An Array is a DATA step construct made up of a group of variables you want to process together or treat similarly. It exists only for the duration of the DATA step and is referred to by the Array name you assign to it. All the variables that are part of your Array must be of the same type (i.e., either all numeric or all character). Each variable that is part of the Array is called an Array element. The general form of an Array definition that uses numeric variables which are already initialized in the DATA step is as follows:

Array   *your-Array-name*   {*number-of-elements*}   *variable-list* ;

The general form to create an Array of character variables and assign their lengths (if they have not yet been initialized) is as follows:

Array   *your-Array-name*   {*number-of-elements*}   *$length*   *variable-list* ;

Note that "*your-Array-name*" can not be the name of one of your data set variables. Multiple Arrays can be assigned with the same DATA step, so take care to name your Arrays in a meaningful way.

Arrays are powerful tools that can help you improve program efficiency and maintainability in many ways. For example, Arrays allow you to:

1. Efficiently manipulate groups of variables by writing one code segment and having it function across multiple variables (e.g., executing logical tests, performing data transformations, etc.)
2. Improve program flexibility when you don't know all the variables you want to analyze at the start of the analysis (see the section on Explicit vs Implicit Arrays later in the paper)
3. Rapidly perform table lookups (see the section on _TEMPORARY_ Arrays later in the paper)
4. Rapidly summarize data in the DATA step resulting in sorted output data w/o having to use PROC SORT (also covered in the section on _TEMPORARY_ Arrays).

Before getting too much further into the topic of Arrays, let's look at a specific example and use it as a starting reference point to discuss details, features and the functionality of Arrays. The code snippet and data example that follows will be our jumping off point.

The below DATA step creates an Array called "sales" containing 12 elements (variables).

```
1  data new_sales;
2    set mydata.sales_data;
3    array sales{12} sales_01_jan sales_02_feb sales_03_mar
4                    sales_04_apr sales_05_may sales_06_jun
5                    sales_07_jul sales_08_aug sales_09_sep
6                    sales_10_oct sales_11_nov sales_12_dec;
7  run;
```

Lines 3-6 above define the Array named "sales". The curly brackets on line 3 are used to assign the number of elements in the Array. In this case, since we have one variable containing sales data for each month of the year, we know we have 12 elements. All the variables on lines 3-6 then become a part of the "sales" Array. The order of the variables you list in the Array assignment matters. The first variable, sales_01_jan, becomes Array element 1, referred to in code as sales{1}, the second variable, sales_02_feb, becomes Array element 2, referred to in code as sales{2}, and so on. So, keep in mind that when referring to the variables in the Array, you will refer to them by their element number rather than their actual name (although you can also refer to individual variable names as needed as well). An example of the data for one household from the "sales" data set referenced above is displayed below in Table 1.

Table 1 – Sales Data for Household 0001

HH_ID	DEPT	SALES_01_JAN	SALES_02_FEB	SALES_03_MAR	…	SALES_12_DEC
0001	1	.	$ 74.25	.	…	.
0001	2	.	.	.	…	$ (164.99)
0001	3	.	.	$ 16.98	…	.
0001	4	.	.	.	…	$ 49.99
0001	5	$ 204.96	.	.	…	$ 29.99
0001	6	.	.	.	…	$ 52.49
0001	7	.	.	.	…	$ 81.00
0001	8	.	.	$ 21.00	…	.
0001	9	.	.	$ 111.75	…	.
0001	10	.	.	.	…	$ 126.98

TYPES OF ARRAYS: EXPLICIT vs. IMPLICIT

The "sales" Array defined above is an EXPLICIT Array. This refers to the fact that we knew, a priori, that we wanted these 12 specific variables to be included in our Array. We, therefore, coded {12} elements followed by our list of 12 variables. But you may not always know beforehand how many or which variables to include in an Array. What do you do when you want business rules to drive the analysis, letting the data which meet those rules drive the processing? In this case, you should consider using an IMPLICIT Array.

Using an IMPLICIT Array lets SAS determine how many variables should be included in the Array. The following code shows how our initial example can be modified to become a bit more flexible, allowing SAS to do some of the work for us.

```
1  data new_sales;
2    set mydata.sales_data;
3    array sales{*} sales_:;
4  run;
```

Line 3 now shows we have changed the “12” to an asterisk {*} within the curly brackets. SAS will now count the number of elements in our Array for us. Since we have defined the Array to include all variable names beginning with “sales_” (i.e., the colon is shorthand for “all variable names beginning with…), SAS will count the number of “sales_” variables for us. A word of caution here. You must be careful if your input dataset contains variables whose names begin with “sales_” but which you do not want included in your analysis. If that is the case, drop them with a DROP= dataset option as you read in your data with the “set” statement.

In the authors’ experience, one of the beauties of using IMPLICIT Arrays is that they allow your code to become infinitely more flexible by incorporating a bit of MACRO code. For example, imagine that our code above actually represents lines 531-534 of a 1200 line program instead of lines 1-4 of our simple example. By changing our variable reference above from “sales_:” to “&mysalesvars” and populating this MACRO variable either at the top of the program (i.e., with a %let statement) or “on the fly” letting DATA set meta-data drive its population and creation (i.e., example below), your code becomes much more bullet-proof. This is displayed in the code below:

```
523  proc sql noprint;
524    select distinct name into :mysalesvars separated by ' '
525       from dictionary.columns
526         where strip(upcase(libname))='MYDATA' and
527               strip(upcase(memname))='SALES_DATA' and
528               strip(upcase(name)) like 'SALES_%';
529  quit;
530
531  data new_sales;
532    set mydata.sales_data(keep=hh_id dept sales_:);
533    array sales{*} &mysalesvars.:;
534  run;
```

We now have code that gets it’s information from the dataset itself rather than relying on user input, thus eliminating potential programmer-introduced errors and allowing the code to function across multiple datasets (or even across multiple versions of this same dataset if it changes, for example, by adding additional sales variables going forward).

TYPES OF ARRAYS: _NUMERIC_ vs _CHARACTER_

As mentioned earlier, Arrays must contain variables which are all of the same type, either all numeric or all character. This makes sense because the purpose of using an Array is to perform the same operations across many Array variables. We would not expect to use the INDEX() function on a numeric variable or to use the SUM() function on a character variable. In addition to the shortcut of using IMPLICIT Arrays described above, SAS provides another shortcut for assigning the numeric or character elements of an Array using the _NUMERIC_ and _CHARACTER_ variable lists along with the IMPLICIT Array. Let’s use this concept on our sales_data data set.

The code below demonstrates how this can be done.

```
1  data new_sales;
2    set mydata.sales_data;
3    array sales{*} _numeric_;
4  run;
```

Note that we are again using the IMPLICIT way of determining the number of Array elements. We are then including all of the NUMERIC variables in the “sales” Array. Is this appropriate? As is always the case, responsibility lies with the carpenter to ensure that the tools are sharp and used correctly.

From the results of the CONTENTS Procedure (see Table 2 below), we can see that in addition to our numeric sales variables, we also have HH_ID which is numeric. If we use the Array to perform any transformations or data derivations, we run the risk of corrupting our household identifier or introducing bias into any statistics derived from our sales variables. What to do, what to do? If we do not need HH_ID for the analysis in question, we could drop the variable from the dataset as we read it in. If we are interested in simply summarizing quarterly or yearly sales data or looking at averages across months, this would work. If, however, we are looking at specific segments of the population or trending household sales over time, we need to keep HH_ID on our data set.

Table 2 – Variable information for MYDATA.SALES_DATA data set

```
Alphabetic List of Variables and Attributes

 #    Variable        Type    Len    Format

 2    DEPT            Char      8
 1    HH_ID           Num       8    22.
 3    SALES_01_JAN    Num       5
 4    SALES_02_FEB    Num       5
 5    SALES_03_MAR    Num       5
 6    SALES_04_APR    Num       5
 7    SALES_05_MAY    Num       5
 8    SALES_06_JUN    Num       5
 9    SALES_07_JUL    Num       5
10    SALES_08_AUG    Num       5
11    SALES_09_SEP    Num       5
12    SALES_10_OCT    Num       5
13    SALES_11_NOV    Num       5
14    SALES_12_DEC    Num       5
```

Another option for solving this problem is to recode the HH_ID variable as character to prevent it’s inclusion in our NUMERIC Array. Since we cannot reassign the TYPE attribute of an existing variable when reading it in from a dataset, the code below shows one work-around for this problem.

```
1  data new_sales(rename=(new_hh_id=hh_id) drop=i);
2    set mydata.sales_data;
3    new_hh_id=put(hh_id,25.);
4    drop hh_id;
5    array sales{*} _numeric_;
6  run;
```

Line 3 creates a new variable NEW_HH_ID which is defined to be $25. Line 4 drops the old HH_ID variable prior to the assignment of the “sales” Array and Line 1 renames the NEW_HH_ID variable as it is being output to the new dataset. Our mydata.sales_data dataset contains only one variable which is character, so we will not load it into an Array, but the concept would work the same, replacing the _numeric_ keyword with the _character_ (or _char_) keyword. We now have a way of placing all our numeric or character variables into an Array even when our dataset contains some variables of our selected type which we do not want to include in the Array.

OK, NOW WHAT?

We now have a basic understanding of how to define a simple, single-dimensional, Array (later in the paper we will discuss use of multi-dimensional Arrays). But what can we really do with them? The following code shows a simple example by building on what we have done previously.

```
1  data new_sales;
2    set mydata.sales_data;
3    array sales{*} sales_:;
4    tot_year_sales=sum(of sales{*});
5  run;
```

Line 4 shows that we can now use the variables in the Array to perform summary functions. Using our Array reference we have created a new TOT_YEAR_SALES variable by summing the values of the 12 monthly sales variables. So what? You can do the same thing if you use the below statement to replace line 4 above:

```
tot_year_sales=sum(of sales_:);
```

True. But, say you want to do the following:

1. recode all your missing values to 0 for building a statistical model
2. change negative sales values (i.e., returns) to 0
3. sum quarterly and yearly variables for use as predictors in your model

You could do this the "old" way using the following code (results for one household are displayed in Table 3 below):

```
data new_sales;
  set mydata.sales_data;
  if sales_01_jan < 0 then sales_01_jan = 0;
  if sales_02_feb < 0 then sales_02_feb = 0;
  if sales_03_mar < 0 then sales_03_mar = 0;
  if sales_04_apr < 0 then sales_04_apr = 0;
  if sales_05_may < 0 then sales_05_may = 0;
  if sales_06_jun < 0 then sales_06_jun = 0;
  if sales_07_jul < 0 then sales_07_jul = 0;
  if sales_08_aug < 0 then sales_08_aug = 0;
  if sales_09_sep < 0 then sales_09_sep = 0;
  if sales_10_oct < 0 then sales_10_oct = 0;
  if sales_11_nov < 0 then sales_11_nov = 0;
  if sales_12_dec < 0 then sales_12_dec = 0;
  q1_sales=sum(of sales_01_jan sales_02_feb sales_03_mar);
  q2_sales=sum(of sales_04_apr sales_05_may sales_06_jun);
  q3_sales=sum(of sales_07_jul sales_08_aug sales_09_sep);
  q4_sales=sum(of sales_10_oct sales_11_nov sales_12_dec);
  tot_year_sales=sum(of sales_:);
run;
```

This is very repetitive coding, not fun to write and, more importantly, it is error-prone and not easily maintainable. In cases where we have a growing number of variables, this quickly becomes an unwieldy way to process data, especially since, in a constantly changing business environment, we may not have a complete list of variables at the start of our analysis. A much better way to produce the same results would be to use some of the basic techniques we've learned about Array processing and to build on the basics by integrating DO-LOOP processing to eliminate repetitive coding. This also allows us to introduce several additional Array topics which will be fully explained shortly (including the DIM() function) and will serve as the building block on which we will begin our discussion of multidimensional Arrays.

The following code demonstrates these techniques and is followed by Table 3, which shows the data which are produced by the code.

```
 1  data new_sales;
 2    set mydata.sales_data;
 3    array sales{*} sales_:;
 4    do i=1 to dim(sales);
 5      if sales{i} < 0 then sales{i}=0;
 6    end;
 7    q1_sales=sum(of sales{1} sales{2} sales{3});
 8    q2_sales=sum(of sales{4} sales{5} sales{6});
 9    q3_sales=sum(of sales{7} sales{8} sales{9});
10    q4_sales=sum(of sales{10} sales{11} sales{12});
11    tot_year_sales=sum(of sales{*});
12  run;
```

Line 3 shows our familiar Array assignment.

Lines 4-6 demonstrate our DO-LOOP processing. Since we want to reassign any negative or missing sales values to 0, and our Array contains all of our sales_01_jan through sales_12_dec variables which we refer to in our Array as sales{1}, sales{2}, etc., we can simply place the single line of code inside a do-loop, replacing the explict references to our Array elements (i.e., 1-12) with the values of the iterating do-loop variable i. The DIM() function returns the number of Array elements in a one-dimensional Array. The DIM() function is often used in conjunction with the {*} when we declare an Array for which we do not explicitly know the number of dimensions beforehand. So line 4 above effectively says “do i=1 to the number of elements in the sales Array” which is 12 in this example.

Lines 7-11 demonstrate one way to create our quarterly and yearly sales variables. The next section, introducing multidimensional Arrays will demonstrate another way we can do this using this new programming construct.

Table 3 – Summarized NEW_SALES Data for Household 0001

HH_ID	DEPT	SALES_01_JAN	SALES_02_FEB	SALES_03_MAR	...	SALES_12_DEC	Q1_SALES	Q2_SALES	Q3_SALES	Q4_SALES	TOT_YEAR_SALES
0001	1	$ -	$ 74.25	$ -	...	$ -	$ 74.25	$ -	$ 61.97	$ -	$ 136.22
0001	2	$ -	$ -	$ -	...	$ -	$ -	$ -	$ -	$ 164.99	$ 164.99
0001	3	$ -	$ -	$ 16.98	...	$ -	$ 16.98	$ -	$ 16.87	$ 28.05	$ 61.90
0001	4	$ -	$ -	$ -	...	$ 49.99	$ -	$ 11.25	$ -	$ 49.99	$ 61.24
0001	5	$ 204.96	$ -	$ -	...	$ 29.99	$ 204.96	$ 43.99	$ -	$ 29.99	$ 278.94
0001	6	$ -	$ -	$ -	...	$ 52.49	$ -	$ 74.99	$ -	$ 52.49	$ 127.48
0001	7	$ -	$ -	$ -	...	$ 81.00	$ -	$ 24.50	$ -	$ 81.00	$ 105.50
0001	8	$ -	$ -	$ 21.00	...	$ -	$ 21.00	$ -	$ -	$ -	$ 21.00
0001	9	$ -	$ -	$ 111.75	...	$ -	$ 111.75	$ -	$ 34.99	$ -	$ 146.74
0001	10	$ -	$ -	$ -	...	$ 126.98	$ -	$ -	$ 149.99	$ 306.95	$ 456.94

Here is an appropriate time to call out the topics of Array bounds and do-loop indexing; in other words, defining the upper and lower bounds of an Array and how we refer to them in do-loop processing. Note that lines 4-6 in the code above refer to variables based on their “position” or reference order in the sales Array. So here, they are referred to using the do-loop incrementing values of 1-12. Although we are defining our sales Array implicitly (e.g., using the {*} to ask SAS to determine how many elements it contains), we could say that the lower bound of the sales Array is 1 and the upper bound is 12. Conforming to this convention, however, is not always useful and, luckily, it is not a requirement. For example, if we want to analyze variables containing yearly sales data for the ten years 2000-2009, we could name our variables sales_2000, sales_2001, sales_2002, etc. We could then use the following code to reference and process our yearly sales variables:

```
array yrsales{2000:2009} sales_2000-sales_2009;
do i=2000 to 2009;
  if yrsales{i} < 0 then yrsales{i}=0;
end;
```

This example was designed to clearly show that the yrsales Array has ten elements, a lower bound of 2000 and an upper bound of 2009. Line 2 of the above code could also be written this way:

```
            do i=lbound(yrsales) to hbound(yrsales);
```

The LBOUND() function in this example returns the value 2000 and the HBOUND() function returns the value 2009, thus improving our code to be more flexible and less error-prone. Note the difference here between the HBOUND() function and the DIM() function. Recall that the DIM() function returns the number of elements in the Array. In the above example, the DIM() function returns a value of 10. Had we used DIM instead of HBOUND, our “do loop” would have attempted to execute the following:

```
      do i=2000 to 10;
```

Although no ERRORs or WARNINGs would be issued to our .log file, this is not at all what we wanted - substituting the above for line 2 in our code example does not result in our missing and negative values being recoded to 0. The moral of this story is that when explicitly defining our Array bounds as shown, we need to be particularly careful in processing our Array references using do-loop processing.

MULTIDIMENSIONAL ARRAYS
Rather than referring to our sales_01_jan through sales_12_dec variables as elements 1-12 of our sales Array, we notice that these 12 variables group themselves nicely into four groups of three variables, which we can use to derive quarterly sales statistics (i.e., January-March, April-June, July-September and October-December). SAS provides us an Array feature that allows us to do this very effectively, while at the same time improving the flexibility of our code. This is the multidimensional Array.

When thinking of our data in terms of a multidimensional Array, it will help to think of our 12 sales variables as a table of 4 rows and 3 columns, as shown in Table 4 below. We can fill in this conceptual model with the renamed sales variables (renamed for purposes described shortly). Keep in mind that this is only a conceptual model and nothing about the way the data are stored is actually changed. Also, note that this tabular multidimensional Array structure exists for each household X department combination in our data set.

Table 4 4 Rows X 3 Columns

Columns

		1	2	3
	1	sales_1	sales_2	sales_3
Rows	2	sales_4	sales_5	sales_6
	3	sales_7	sales_8	sales_9
	4	sales_10	sales_11	sales_12

In the table above, each row corresponds to one of our calendar quarters and each column corresponds to a calendar month within that quarter. While our data for each household X department combination is still stored on a single record in our data set, our updated code loads our data (arranged conceptually this way) into a multidimensional Array which SAS can use to manipulate and summarize using similar syntax to the single dimensional Arrays we saw earlier. The code below shows how to do this (a line-by-line explanation follows).

```
1   data new_sales;
2   set mydata.sales_data(rename=(sales_01_jan=sales_1
3                                 sales_02_feb=sales_2
4                                 <...rename Mar-Oct variables...>
5                                 sales_11_nov=sales_11
6                                 sales_12_dec=sales_12));
7     array sales_qtrs{4,3} sales_1-sales_12;
8     array sales_q_sum{4} sales_q_sum1-sales_q_sum4;
9     do i=1 to 4;
```

```
10      do j=1 to 3;
11        if sales_qtrs{i,j} < 0 then sales_qtrs{i,j}=0;
12        sales_q_sum{i}=sum(of sales_q_sum{i} sales_qtrs{i,j});
13      end;
14      total_year_sales=sum(of total_year_sales sales_q_sum{i});
15    end;
16  run;
```

Line 1 creates our new_sales dataset in the work library.

Lines 2-6 read in mydata.sales_data and rename all the sales variables as they are read in. This will allow us to use a short-hand notation when referencing groups of variables to define our Arrays. For example…

Line 7 Defines the sales_qtrs Multidimensional Array using the short-hand of "sales_1-sales_12" instead of typing out all 12 variables to define the Array. This is where our conceptual representation of our 4 Rows by 3 Columns table becomes most useful. For each household X department combination, the values of the sales_1-sales_12 variables all lie in a single observation on our SAS dataset. But the sales_qtrs Multidimensional Array is assigned to refer to them *as if* they were arranged in this tabular format. The first coordinate in the Array assignment, {4}, indicates how many rows of our conceptual table SAS will reference and the second coordinate in the Array assignment, {3}, indicates how many columns SAS will reference in the Array. So, referring to Table 4 again, if we want to refer to the middle cell of row 3 (corresponding to Calendar Quarter 3, Month 2, aka August), we would refer to it as sales_qtrs{3,2}. Similarly, December would be referred to as sales_qtrs{4,3}. This notation will become supremely useful when we place references to our Array inside the nested do-loops seen in lines 9-15.

[Note: The SAS documentation starts this explanation by describing the right-most coordinate as the number of columns with the next coordinate working your way left indicating the number of rows. Since Arrays can have more than two dimensions, each additional column working your way left represents a higher-level dimension. That said, the topic of higher-order Multidimensional Arrays quickly becomes out-of-scope for this paper and we will leave exploration of this topic to the reader – and future papers.]

Line 8 does two things. First, it creates a one dimensional Array which we assign to have 4 elements. Secondly, in initializing this Array, we actually are creating 4 numeric variables, sales_q_sum1, sales_q_sum2, sales_q_sum3 and sales_q_sum4. In the following lines, these will be populated with quarterly sales summaries.

Line 9 (`do i=1 to 4;`) corresponds to the first dimension in our Array, the number of rows in our conceptual table. Everything we code between the do and the end block here will be done once for each row in our conceptual table.

Line 10 (`do i=1 to 3;`) corresponds to the second dimension in our Array, the number of columns in our conceptual table. Everything we code within this do and end block will be done once for each column in our conceptual table. Since we have two dimensions, we now have a way to reference each individual cell of our conceptual table using a set of coordinates which will be iterated / incremented through use of our two do-loops.

Lines 11-12 Since these two code lines lie within the inner-most do-loop, we will refer to variables here using both dimensional coordinates. For example, line 11 is where we have moved our reassignment of missing and negative values for all 12 sales variables to 0. Line 12 takes advantage of the fact that each column will be processed once (i.e., do j=1 to 3;) for each row (do i=1 to 4;) before hitting the implicit output statement when SAS reaches the run statement to end the DATA step. We can use this fact, and the SUM function, to derive the quarterly sales variables sales_q_sum1-sales_q_sum4.

Line 13 ends our inner do-loop.

Line 14 sums our quarterly sales variables to create a total yearly sales variable. Because it is inside our "row-level" do-loop (i.e., do i=1 to 4), we use it to sum our quarterly sales variables.

We can now see that with just a little creative application of our basic Array techniques, we can perform complicated data summaries with minimal programming effort. And this serves as a convenient transition into our next topic, using _TEMPORARY_ Arrays to rapidly summarize data.

WHAT IS A _TEMPORARY_ ARRAY?
The general form of a _TEMPORARY_ Array definition that stores numeric data is as follows:

Array *your-Array-name* {number-of-elements} _temporary_ (initial-values);

The general form of a _temporary_ Array definition that stores character data is as follows:

Array *your-Array-name* {number-of-elements} $ length _temporary_ (initial-values);

This defines a set of array elements that differ primarily in two ways from other array variables. First, the values stored in the array elements are automatically retained. Secondly, the array elements can only be identified by explicit array reference, because the array elements do not have names. Without names, the values contained in the _temporary_ array do not appear in the output data set.

So, what use is such an unusual data structure? The _temporary_ array is especially designed to hold sets of constants used for computations. For example, suppose a financial institution supports five types of interest-bearing accounts, each with a different interest rate. An example of the “bank_customers” SAS data set comprised of accountholders along with their average balance for a period of time is displayed below in Table 5.

Table 5 – Average Balances for Five Types of Interest-Bearing Accounts

ACCT_NO	ACCT_TYPE	ACCT_INDEX	AVG_BALANCE
KAFT-2459	CK2	2	$720.92
GMKB-5165	CK2	2	$6,014.30
VWHR-8769	SV1	3	$197.21
HPWB-7143	CK1	1	$4,825.06
XPIG-4341	SV2	4	$265.27
RIKP-8125	SV1	3	$2,053.86
VLAK-1804	CK1	1	$8,837.14
EHYE-1515	CK1	1	$4,388.91

The interest amount for each account can be computed with the following SAS logic:

```
1  data interest_accrual;
2     array int{5} _temporary_ (0.0125 0.0135 0.0150 0.0165 0.0168);
3     set bank_customers;
4     interest_amount=avg_balance*int{acct_index};
5  run;
```

Line 2 defines the _temporary_ array which stores the five interest rates for the different types of accounts. In this simple example, each interest rate is indexed by the account index field. Recall from our earlier discussion that we reference an Array element based on it’s position in the Array, so the second value in the _temporary_ Array, 0.0135, will be referenced by an index value of 2.

Line 4 computes the interest amount by using “account_index” to reference the appropriate interest rate stored in the _temporary_ array. Note that the interest rate used in the computation will not be saved to the output data set since it has not been assigned to a data step variable.

Since there are only a few interest rates in this example, this could have been accomplished with a series of “if-then” statements or a “select-when” structure, based on the account type or account index. The data step would have more lines of code, but would work just fine. The benefits of the _temporary_ array structure are more apparent with a larger number of constants.

In many real-world situations, however, the number of constants needed for computations exceeds what is practical for "if-then" statements. In these situations, the list of constants typically exists in a separate data set. If the primary input data set is not terribly large, sorting both the constants data set and primary input data set and then merging them together to attach the appropriate constants is not impractical. But often in business, we encounter data sets with millions of records that require hundreds, if not thousands, of constants for computation. In these cases, "if-then" statements or "select-when" structures are not viable options. Sorting and merging may even strain the limits of temporary space and/or CPU resources, let alone processing time. In these cases, loading a _temporary_ array during the first pass of the data step can be an efficient approach. The basic SAS logic is as follows:

```
data out_file;
  if _n_=1 then
  do until(done);
     array constants{number-of-elements} _temporary_;
     set constants_list end=done;
     [assign values to array elements]
  end;
  set primary_input_file;
  [computations using _temporary_ array elements]
run;
```

Also, keep in mind that this assumes there is a set of integers that uniquely connects the list of constants to the primary input data as the way of indexing the array elements. Next we will take this concept to the next level of complexity with many unique values for our constants.

USING _TEMPORARY_ ARRAYS FOR SUMMARIZING

The features of the _temporary_ array can be used to perform non-trivial summarization with just a single data step. Since the _temporary_ array elements are always automatically retained, the idea is to accumulate values within the _temporary_ elements as the input data set is being read. Then once all records have been read, the summarized values stored in the array can be output.

For example, suppose a company supplies a few thousand different products (numbered 1000 to 3999) for millions of purchasers. In this example, each purchaser buys no more than 50 different items at a time. The SAS data set, work_orders, is comprised of purchase orders as shown in Table 6 below.

Table 6 – Purchase Orders

CUST_ID	ORDER_ID	NO_PRODUCTS	PROD1	PROD2	…	PROD50	UNITS1	UNITS2	…	UNITS50	SALES1	SALES2	…	SALES50
YXP-850	LTAI-6999	2	2783	3599	…	.	4	1	…	.	$394.96	$68.41	…	.
RVU-992	WDJI-2774	1	3321	.	…	.	8	.	…	.	$731.12	.	…	.
VNI-183	DSIT-7681	50	2278	2787	…	2973	5	1	…	5	$142.05	$19.23	…	$172.85
FTT-660	IAGB-9162	15	1595	2999	…	.	1	6	…	.	$71.49	$370.08	…	.
GBC-283	OQJX-9041	20	1321	1879	…	.	2	6	…	.	$51.48	$512.28	…	.
MTM-499	RJFP-5724	6	2848	2436	…	.	7	6	…	.	$24.64	$90.06	…	.
JYM-852	VBBL-7558	12	3576	1962	…	.	7	5	…	.	$254.66	$41.55	…	.
DGG-052	EBKG-3944	25	3542	3318	…	.	6	2	…	.	$488.58	$160.36	…	.
LHJ-054	JABS-2284	30	1454	1703	…	.	2	5	…	.	$149.80	$481.40	…	.
WSG-495	QLEC-4809	50	1314	3597	…	2701	8	8	…	8	$227.04	$353.28	…	$83.12

If we want to produce a summary of number of purchase occasions, number of units and dollar sales by product number there are several approaches that can be taken. One approach is to transpose the entire data set to cust_id/order_id/product_number level, sort by product_number and then summarize using proc means (or proc summary) with "by-processing." But, if there are millions of cust_id records, the transpose, sort and summary will take a long, long time and perhaps more space than your system provides.

Another approach is to loop 50 times through the original data set performing a sort and summary on each of the 50 item fields. But, this isn't terribly efficient either.

The following data step produces a summarized data set with only one pass through the input data set, with no sort and no proc means (proc summary).

```
 1 data summarized;
 2    keep product_number purchase_orders units sales;
 3    array order{3,50} prod1-prod50 units1-units50 sales1-sales50;
 4    array tally{3,1000:3999} _temporary_;
 5       ** dim1 - 1-total orders, 2-total units, 3-total sales **;
 6       ** dim2 - 1000:3999 refers to product numbers **;
 7    set work_orders end=done;
 8    do i=1 to no_products;
 9       product_id=order{1,i};
10       tally{1,product_id}=sum(1,tally{1,product_id});
11       tally{2,product_id}=sum(order{2,i},tally{2,product_id});
12       tally{3,product_id}=sum(order{3,i},tally{3,product_id});
13    end;
14    if done then
15    do product_number=1000 to 3999;
16       purchase_orders=tally{1,product_number};
17       if purchase_orders ^= . then do;
18          units=tally{2,product_number};
19          sales=tally{3,product_number};
20          output;
21       end;
22    end;
23 run;
```

Line 2 specifies the few summarized fields for the output data set. At most there will be a record for product number 1000, 1001, 1002, and so on, up to product number 3999.

Line 3 specifies an array similar to that in our first quarterly sales example. It organizes 150 (=3x50) of the fields in each input record.

Line 4 specifies the _temporary_ array, named "tally" that will store our three different sums (number of orders, sum of units, and sum of sales) for every different product number found in the entire input data set. Note that the second dimension index has been specified as values from 1000 to 3999. As we discussed previously, Array indices may be specified for any range of integers, including zero and negatives.

Lines 5 and 6 are comments to document the array structure.

Line 7 reads the input data set. The "end=" data set option assigns a value of 1 to the variable "done" once the last record has been read. The value of "done" is zero otherwise. The output data set will only be created after all input records have been read and processed (see line 14).

Line 8 begins the loop to scan across the input record. In this example we have the luxury of already knowing just how many of the 50 possible items have been purchased as the value of "no_products."

Line 9 assigns the i^{th} unique purchased product number to the field named product_id.

Line 10 increments the count of orders containing the given product_id by 1.

Line 11 adds the number of units of product_id in this purchase order to the total being held in the "tally" array. Note that order{2,i} refers to the units purchased for the i^{th} product_id in the input record.

Line 12 adds the sales of product_id in this purchase order to the total being held in the “tally” array. Note that order{3,i} refers to the sales for the ith product_id in the input record.

Line 14 checks the value of “done”. Note that “if done” yields the same result as “if done=1”. Recall, that the output file is only generated once the entire input file has been read and processed.

Line 15 tells SAS to begin looping through all the valid product numbers for the company, 1000 through 3999. This loop is closed on line 22.

Line 16 assigns the number of purchase orders counted and stored in the “tally” array for a given product_number. If none were purchased, this field will be missing.

Line 17 checks the value assigned in Line 16. If the value is not missing, then purchases were made of this item and we want to output the sums for this item.

Line 18 assigns the sum of units for this item, which we know has sold, to the field “units” on the output file. Likewise, Line 19 assigns the sum of sales totaled for this item to the field “sales” on the output file.

Line 20 indicates to output the values of the fields listed in the “keep” statement. Note that this is the only point in the data step in which output is generated.

The output SAS data set would look something like that in Table 7 below.

Table 7 – Purchase Orders Summarized by Product Number

PRODUCT_NUMBER	PURCHASE_ORDERS	UNITS	SALES
1000	570	3,990	$250,811.40
1001	1,231	1,231	$77,774.58
1003	1,495	11,960	$780,748.80
1004	728	1,456	$44,073.12
1005	576	1,152	$47,485.44
:	:	:	:
3997	783	4,698	$49,610.88
3999	817	5,719	$161,047.04

We now have summaries for each product number that was sold, listed in product_id order. All this was accomplished with one pass through the input data set without sorting. Note that in this example, products numbered 1002 and 3998 are not listed in the output since there were no sales of these items.

WHAT ABOUT HASH TABLES?

The functionality demonstrated by the _temporary_ arrays in these examples could have been accomplished by means of Hash Tables instead. The question is when would one use a _temporary_ array instead of a Hash Table?

As noted earlier, the use of arrays is predicated on integer indexing of the array elements. Hash tables, on the other hand, can reference data based on complex combinations of numeric and character data. If there is no means of integer indexing, then an array structure cannot be used and Hash tables may prove to be a better option. However, if there is a means of indexing the information by a set of integers, then the number of elements needed may determine if a _temporary_ array is a good option. The limit of what is reasonable depends on the computing resources available. Generally, several hundred array elements should not be problem. The authors have routinely used _temporary_ arrays consisting of several thousand elements.

CONCLUSION

There are many things a programmer does on a regular basis which we would call “repeated processing”, meaning we perform the same operation across many variables often over and over again. Array processing can make this type of work much more efficient. Adding some basic knowledge of this Data step construct to your SAS Programming toolbox will help you become better at what you do. We hope you have gained enough knowledge to begin exploring the use of Arrays in your everyday coding. Additionally, through some of our more advanced examples, we hope we have sparked your curiosity enough to explore the depths of all that is possible using these powerful programming constructs.

REFERENCES

SAS OnlineDoc® 9.1.3. SAS Institute Inc., Cary, NC, USA.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the authors below:

Joe Novotny
dunnhumby USA
444 W. 3rd Street
Cincinnati, OH 45202
joe.novotny@us.dunnhumby.com

Paul Kollner
dunnhumby USA
444 W. 3rd Street
Cincinnati, OH 45202
paul.kollner@us.dunnhumby.com

AUTHOR BIOGRAPHIES

Joe Novotny began programming SAS during graduate school at Penn State in 1996. After receiving his Master's in Human Development, he switched industries to pharmaceutical research where he used SAS for 8 years analyzing phase II and III clinical trials. In 2007, Joe switched industries again and began working for dunnhumbyUSA as a Direct Marketing Analyst. Joe has presented at MWSUG and NESUG and is currently involved with the Cincinnati SAS User's Group (CinSUG) and MWSUG.

Paul Kollner has been with dunnhumbyUSA for 3½ years as a Direct Marketing Analyst and has been in market research since 2001, after 11 years as an actuarial analyst. Paul first learned SAS through university courses in the late 1980's and has been programming in SAS since the early 1990's. The need to manipulate large data sets and perform unique analyses has driven Paul's desire to move past SAS basics into more advanced areas like arrays.