Paper 242-30

Arrays Made Easy: An Introduction to Arrays and Array Processing

Steve First and Teresa Schudrowitz, Systems Seminar Consultants, Inc., Madison, WI

ABSTRACT

Many programmers often find the thought of using arrays in their programs to be a daunting task, and as a result they often shy away from arrays in their code in favor of better-understood, but more complex solutions. A SAS array is a convenient way of temporarily identifying a group of variables for processing within a data step. Once the array has been defined the programmer is now able to perform the same tasks for a series of related variables, the array elements. Once the basics of array processing are understood arrays are a simple solution to many program scenarios.

Throughout this paper we will review the following array topics:

1) Why do we need arrays?
2) Basic array concepts
   a) Definition
   b) Elements
   c) Syntax
   d) Rules
3) Using array indexes
4) One dimension arrays
5) Multi-dimension arrays
6) Temporary arrays
7) Explicit vs. implicit subscripting
8) Sorting arrays
9) When to use arrays
10) Common errors and misunderstandings

INTRODUCTION

Most mathematical and computer languages have some notation for repeating or other related values. These repeated structures are often called a matrix, a vector, a dimension, a table, or in the SAS data step, this structure is called an array. While every memory address in a computer is an array of sorts, the SAS definition is a group of related variables that are already defined in a data step. Some differences between SAS arrays and those of other languages are that SAS array elements don't need to be contiguous, the same length, or even related at all. All elements must be character or numeric.

WHY DO WE NEED ARRAYS?

The use of arrays may allow us to simplify our processing. We can use arrays to help read and analyze repetitive data with a minimum of coding.

An array and a loop can make the program smaller. For example, suppose we have a file where each record contains 24 values with the temperatures for each hour of the day. These temperatures are in Fahrenheit and we need to convert them to 24 Celsius values. Without arrays we need to repeat the same calculation for all 24 temperature variables:

```
data;
 input etc.
 celsius_temp1 = 5/9(temp1 – 32);
 celsius_temp2 = 5/9(temp2 – 32);
 . . .
 celsius_temp24 = 5/9(temp24 – 32);
run;
```

An alternative is to define arrays and use a loop to process the calculation for all variables:

```
data;
 input etc.
 array temperature_array {24} temp1-temp24;
 array celsius_array {24} celsius_temp1-celsius_temp24;
 do i = 1 to 24;
  celsius_array{i} = 5/9(temperature_array{i} – 32);
 end;
run;
```

While in this example there are only 24 elements in each array, it would work just as well with hundreds of elements. In addition to simplifying the calculations, by defining arrays for the temperature values we could also have used them in the input statement to simplify the input process. It should also be noted, while TEMP1 is equivalent to the first element, TEMP2 to the second etc., the variables do not need to be named consecutively. The array would work just as well with non-consecutive variable names.

```
array sample_array {5} x a i r d;
```

In this example, the variable x is equivalent to the first element, a to the second etc.

Arrays may also be used to provide table lookups. For instance, we have different percentage rates that will be applied to a representative's sales amounts to determine their commission. The percentage amount may be stored within an array structure, and then the array could provide a location to look up the appropriate percentage for inclusion in the commission calculation.

BASIC ARRAY CONCEPTS

Arrays within SAS are different than arrays in other languages. SAS arrays are another way to temporarily group and refer to SAS variables. A SAS array is not a new data structure, the array name is not a variable, and arrays do not define additional variables. Rather, a SAS array provides a different name to reference a group of variables.

The ARRAY statement defines variables to be processed as a group. The variables referenced by the array are called elements. Once an array is defined, the array name and an index reference the elements of the array. Since similar processing is generally completed on the array elements, references to the array are usually found within DO groups.

ARRAY STATEMENT

The statement used to define an array is the ARRAY statement.

```
array array-name {n}  <$> <length> array-elements <(initial-values)>;
```

The ARRAY statement is a compiler statement within the data step. In addition, the array elements cannot be used in compiler statements such as DROP or KEEP. An array must be defined within the data step prior to being referenced and if an array is referenced without first being defined, an error will occur. Defining an array within one data step and referencing the array within another data step will also cause errors, because arrays exist only for the duration of the data step in which they are defined. Therefore, it is necessary to define an array within every data step where the array will be referenced

The ARRAY statements provides the following information about the SAS array:

- array-name – Any valid SAS name
- n – Number of elements within the array
- $ - Indicates the elements within the array are character type variables
- length – A common length for the array elements
- elements – List of SAS variables to be part of the array
- initial values – Provides the initial values for each of the array elements

The array name will be used as part of the array reference when the array elements are used in processing. The name must follow the same rules as variable names, therefore, any valid SAS name is a valid array name. When naming an array it is best to avoid using an array name that is the same as a function name to avoid confusion. While parentheses or square brackets can be used when referencing array elements, the braces {} are used most often since they are not used in other SAS statements. SAS does place one restriction on the name of the array. The array name may not be the same name as any variable on the SAS data set.

The elements for an array must be all numeric or all character. When the elements are character it is necessary to indicate this at the time the array is defined by including the dollar sign ($) on the array statement after the reference to the number of elements. If the dollar sign is not included on the array statement, the array is assumed to be numeric.

When all numeric or all character variables in the data set are to be elements within the array, there are several special variables that may used instead of listing the individual variables as elements. The special variables are:

NUMERIC	- when all the numeric variables will be used as elements
CHARACTER	- when all the character variables will be used as elements
ALL	- when all variables on the data set will be used as elements and the variables are all the same type

N is the array subscript in the array definition and it refers to the number of elements within the array. A numeric constant, a variable whose value is a number, a numeric SAS expression, or an asterisk (*) may be used as the subscript. The subscript must be enclosed within braces {}, square brackets [], or parentheses (). In our temperature example the subscript will be 24 for each of the 24 temperature variables:

```
array temperature_array {24} temp1 – temp24;
```

When the asterisk is used, it is not necessary to know how many elements are contained within the array. SAS will count the number of elements for you. An example of using the asterisk is when one of the special variables defines the elements.

```
array allnums {*} _numeric_;
```

When it is necessary to know how many elements are in the array, the DIM function can be used to return the count of elements.

```
do i = 1 to dim(allnums);
  allnums{i} = round(allnums{i},.1);
end;
```

In this example, when the array ALLNUMS is defined, SAS will count the number of numeric variables used as elements of the array. Then, in the DO group processing, the DIM function will return the count value as the ending range for the loop.

ARRAY REFERENCES

When an array is defined with the ARRAY statement SAS creates an array reference. The array reference is in the following form:

```
array-name{n}
```

The value of n will be the element's position within the array. For example, in the temperature array defined above the temperature for 1:00 PM is in the variable TEMP13. The array element has been assigned the 13th position within the array. Therefore, the array reference will be:

```
temperature_array{13}
```

The variable name and the array reference are interchangeable. When an array has been defined in a data step either the variable name or the array reference may be used.

Variable Name	Array Reference
temp1	temperature_array{1}
temp2	temperature_array{2}
temp3	temperature_array{3}
temp4	temperature_array{4}
temp5	temperature_array{5}

An array reference may be used within the data step in almost any place other SAS variables may be used including as an argument to many SAS functions. If the data step does not have an ARRAY statement to define the array and create the array reference, errors will occur. When an array is referenced within a data step, it must be defined with an ARRAY statement in the same data step.

USING ARRAY INDEXES

The array index is the range of array elements. In our temperature example, we are looking at temperatures for each of the 24 hours of the day. The array is defined as:

```
array temperature_array {24} temp1 – temp24;
```

Another variation in SAS arrays from arrays within other languages is, subscripts are 1-based by default where arrays in other languages may be 0-based. When we set the array bounds with the subscript and only specify the number of elements within the array as our upper bound, the lower bound is the default value of 1. In our example, the index begins with the lower bound of 1 and ends with the upper bound of 24.

There may be scenarios when we want the index to begin at a lower bound other than 1. This is possible by modifying the subscript value when the array is defined. For this example we are using our same temperature variables. Only this time we only want the temperatures for the daytime, temperatures 6 through 18. In this example the array is defined as:

```
array temperature_array {6:18} temp6 – temp18;
```

The subscript will be written as the lower bound and upper bound of the range, separated by a colon.
This technique can simplify array usage by using natural values for the index. Examples of this might be to use a person's age, or to use a year value to get to the correct element.

ONE DIMENSION ARRAYS

A simple array may be created when the variables grouped together conceptually appear as a single row. This is known as a one-dimensional array. Within the Program Data Vector the variable structure may be visualized as:

temperature_array	{1}	{2}	{3}	{4}	{5}	…	{24}
Temperature Variables	temp1	Temp2	temp3	temp4	temp5	…	temp24

The array statement to define this one-dimensional array will be:

```
array temperature_array {24} temp1 – temp24;
```

The array has 24 elements for the variables TEMP1 through TEMP24. When the array elements are used within the data step the array name and the element number will reference them. The reference to the ninth element in the temperature array is:

```
temperature_array{9}
```

MULTI-DIMENSION ARRAYS

A more complex array may be created in the form of a multi-dimensional array. Multi-dimensional arrays may be created in two or more dimensions. Conceptually, a two-dimensional array appears as a table with rows and columns. Within the Program Data Vector the variable structure may be visualized as:

2nd Dimension	SALE_ARRAY		{r,1}	{r,2}	{r,3}	{r,4}	…	{r,12}
1st Dimension	Sales Variables	{1,c}	SALES1	SALES2	SALES3	SALES4	…	SALES12
	Expense Variables	{2,c}	EXP1	EXP2	EXP3	EXP4	…	EXP12
	Commission Variables	{3,c}	COMM1	COMM2	COMM3	COMM4	…	COMM12

The array statement to define this two-dimensional array will be:

```
array sale_array {3, 12} sales1-sales12 exp1-exp12 comm1-comm12;
```

The array contains three sets of twelve elements. When the array is defined the number of elements indicates the number of rows (first dimension), and the number of columns (second dimension). The first dimension of the array is the three sets of variable groups: sales, expense, and commission. The second dimension is the 12 values within the

group. When the array elements of a multi-dimensional array are used within the data step the array name and the element number for both dimensions will reference them. The reference to the sixth element for the expense group in the sales array is:

```
sale_array{2,6}
```

Three and more dimensions can be defined as well. It should be noted that if a PROC PRINT is run, only the actual variable names are displayed instead of the array elements, so it can sometimes be difficult to visualize the logical structure.

TEMPORARY ARRAYS

A temporary array is an array that only exists for the duration of the data step where it is defined. A temporary array is useful for storing constant values, which are used in calculations. In a temporary array there are no corresponding variables to identify the array elements. The elements are defined by the key word _TEMPORARY_.

When the key word _TEMPORARY_ is used, a list of temporary data elements is created is created in the Program Data Vector. These elements exist only in the Program Data Vector and are similar to pseudo-variables.

array	{1}	{2}	{3}	{4}	{5}	{6}
Variables						
Values	0.05	0.08	0.12	0.20	0.27	0.35

One method of setting the values for a temporary array's elements is to indicate initial values in the ARRAY statement. Therefore, a temporary array of rate values might be defined as follows:

```
array rate {6} _temporary_ (0.05 0.08 0.12 0.20 0.27 0.35);
```

The values for the temporary data elements are automatically retained across iterations of the data step, but do not appear in the output data sets. With a few exceptions, temporary data elements behave in a manner similar to variables. Temporary data elements do not have names, therefore they may only be referenced using the array reference. The asterisk subscript cannot be used when defining a temporary array and explicit array bounds must be specified for temporary arrays.

We are now able to apply the constant values defined in the array. For example: when a customer is delinquent in payment of their account balance, a penalty is applied. The amount of the penalty depends upon the number of months that the account is delinquent. Without array processing this IF-THEN processing would be required to complete the calculation:

```
if month_delinquent eq 1 then balance = balance + (balance * 0.05);
else if month_delinquent eq 2 then balance = balance + (balance * 0.08);
else if month_delinquent eq 3 then balance = balance + (balance * 0.12);
else if month_delinquent eq 4 then balance = balance + (balance * 0.20);
else if month_delinquent eq 5 then balance = balance + (balance * 0.27);
else if month_delinquent eq 6 then balance = balance + (balance * 0.35);
```

By placing the penalty amounts into a temporary array, the code for calculating the new balance can be simplified. The penalty amounts have been stored in the temporary array RATE. The new account balance with the penalty can now be calculated as:

```
array rate {6} _temporary_ (0.05 0.08 0.12 0.20 0.27 0.35);
if month_delinquent ge 1 and month_delinquent le 6 then
  balance = balance + (balance * rate{month_delinquent});
```

In addition to simplifying the code, the use of the temporary array also improves performance time.

Setting initial values is not required on the ARRAY statement. The values within a temporary array may be set in another manner within the data step.

```
array rateb {6} _temporary_;
do i = 1 to 6;
  rateb{i} = i * 0.5;
end;
```

EXPLICIT VS IMPLICIT SUBSCRIPTING

Earlier versions of SAS originally defined arrays in a more implicit manner as follows:

```
array array-name<(index-variable)> <$> <length> array-elements <(initial-values)>;
```

In an implicit array, an index variable may be indicated after the array name. This differs from the explicit array previously discussed where a constant value or an asterisk, as the subscript, denotes the array bounds. When an implicit array is defined, processing for every element in the array may be completed with a DO-OVER statement. The variable specified as the index variable is tied to the array and should only be used as the array index.

Example:

```
array item(j) $ 12 x1-x12;
do over item;
  put item;
end;
```

When referencing the array, only the array name is specified. Thus, the implied notation, instead of the more explicit mode described earlier. If a program needed to reference by index-variable, that can also be done, but must be specified in a separate statement.

```
array item(j) $ 12 x1-x12;
do j=1 to 12;
  put item;
end;
```

Because of the difficulty understanding the more cryptic implicit arrays, explicit arrays are recommended. Implicit array support was left in SAS only to insure older programs would continue to run.

SORTING ARRAYS

There are several new experimental call routines in SAS 9.1 that can be used to sort across a series of variables. SORTN can be used to sort numeric variables and SORTQ for character fields. An example of sorting several numeric variables is as follows:

```
data _null_;
 array xarry{6} x1-x6;
 set ds1;
 call sortn(of x1-x6);
run;
```

When an observation from ds1 is processed the values brought into the Program Data Vector appear as follows:

xarry	{1}	{2}	{3}	{4}	{5}	{6}
Variables	x1	x2	x3	x4	x5	x6
Values	0.27	0.12	0.20	0.08	0.35	0.05

The SORTN call routine will sort the values of the variables in ascending order and replace the Program Data Vector values with the new sorted values. Thus, after the call routine the Program Data Vector will appear as follows:

xarry	{1}	{2}	{3}	{4}	{5}	{6}
Variables	x1	x2	x3	x4	x5	x6
Values	0.05	0.08	0.12	0.20	0.27	0.35

Because the values for the variables are now different the value selected by the array reference will be affected. For instance, to calculate rate we must add the value in the array reference to 0.75 as follows:

```
rate = 0.75 + xarry{i};
```

If the calculation is completed prior to the SORTN call routine and i is equal to 3, rate would be 0.95. On the other hand, if the same calculation were to be completed after the call routine, rate would be 0.87.

WHEN TO USE ARRAYS

It makes sense to use arrays when there are repetitive values that are related and the programmer needs to iterate though most of them. The combination of arrays and do loops in the data step lend incredible power to programming. The fact that the variables in the array do not need to be related or even contiguous makes them even more convenient to use.

COMMON ERRORS AND MISUNDERSTANDINGS

Common errors and misunderstandings occur in array processing. This section will review several of these and how they are resolved.

INVALID INDEX RANGE

In the processing of array references, it is easy to use an index value from outside the array bounds. A typical instance when this occurs is while looping through the array references with DO group processing.

```
data dailytemp;
  set tempdata;
  array temperature_array {24} temp1-temp24;
  array celsius_array {24} celsius_temp1-celsius_temp24;
  do until (i gt 24);
    i = 1;
    celsius_array{i} = 5 / 9 * (temperature_array{i} – 31);
  end;
  i=0;
  drop i;
run;
```

In the scenario a DO-UNTIL loop is used to process the 24 different temperatures. When a DO-UNTIL loop is processed the evaluation of the expression is not completed until the bottom of the loop. After processing for i equal to 24, the expression is still true. Therefore, processing remains within the loop and i will be incremented to 25. Both arrays used in the calculation were defined with only 24 elements. An index value of 25 is greater than the array's upper bound. As a result, the data error message "Array subscript out of range" will be received.

There are two possible resolutions to this scenario. One possibility is to continue using the DO-UNTIL loop, but change the expression to check for I greater than 23:

```
do until (i gt 23);
  i = 1;
  celsius_array{i} = 5 / 9 * (temperature_array{i} – 31);
end;
```

Another possibility is to modify the loop to DO-WHILE processing. A DO-WHILE loop evaluates the expression first and if the expression is true the processing within the loop will continue. Therefore, the modified code for using a DO-WHILE loop would be:

```
do while (i le 24);
  i = 1;
  celsius_array{i} = 5 / 9 * (temperature_array{i} – 31);
end;
```

FUNCTION NAME AS AN ARRAY NAME

The use of a function name as an array name on the ARRAY statement may cause unpredictable results. If a function name is used as an array name, SAS will no longer recognize the name as a function name. Therefore, rather than seeing parenthetical values as function arguments, SAS now sees those values as the index value for an array reference.

When a function name is used to define an array name SAS will provide a warning in the log. If the function is then used within the data step error messages may be received as result of SAS attempting to interpret the function arguments as an array reference.

```
8    data dailytemp;
9      set tempdata;
10     array mean {24} temp1-temp24;
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.
11     do i = 1 to 24;
12       meantemp = mean(of temp1-temp24);
ERROR: Too many array subscripts specified for array MEAN.
13     end;
14   run;
15
```

Avoiding function names as array names is a simple resolution in this scenario.

ARRAY REFERENCED IN MULTIPLE DATA STEPS, BUT DEFINED IN ONLY ONE

Every data step where an array is referenced it must be defined within the step with an ARRAY statement. A sample program contains the following data steps:

```
data dailytemp;
  set tempdata;
  array temperature_array {24} temp1-temp24;
  array celsius_array {24} celsius_temp1-celsius_temp24;
  do i = 1 to 24;
    celsius_array{i} = 5 / 9 * (temperature_array{i} – 31);
  end;
run;

data celsius;
  set dailytemp;
  do i = 1 to 24;
    if celsius_array{i} lt 0 then tempdesc = ‘below freezing’;
    else if celsius_array{i} gt o then tempdesc = ‘above freezing’;
  end;
run;
```

The first data step contains the definition of the array CELSIUS_ARRAY on the second ARRAY statement. References to the array do not cause any issues. The second data step has two references to the array but the array has not been defined within this data step.

The resolution to this issue requires the inclusion of the ARRAY statement to define the array within every data step where it is referenced. Therefore, the second data step would be modified as follows:

```
data celsius;
  set dailytemp;
  array celsius_array {24} celsius_temp1-celsius_temp24;
  do i = 1 to 24;
    if celsius_array{i} lt 0 then tempdesc = ‘below freezing’;
    else if celsius_array{i} gt o then tempdesc = ‘above freezing’;
  end;
run;
```

CONCLUSION

Arrays are not a mystery, but a wonderful tool for Data step programmers that need to reference repetitive values.

CONTACT INFORMATION

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

Steve First
Systems Seminar Consultants, Inc.
2997 Yarmouth Greenway Dr
Madison, WI 53711
Work Phone: (608) 279-9964 x302
Fax: (608) 278-0065
Email: sfirst@sys-seminar.com
Web: www.sys-seminar.com

Teresa Schudrowitz
Systems Seminar Consultants, Inc.
2997 Yarmouth Greenway Dr
Madison, WI 53711
Work Phone: (608) 279-9964 x309
Fax: (608) 278-0065
Email: tschudrowitz@sys-seminar.com
Web: www.sys-seminar.com