Paper BB-157

Arrays – Data Step Efficiency

Harry Droogendyk, Stratia Consulting Inc., Lynden, ON

ABSTRACT

Arrays are a facility common to many programming languages, useful for programming efficiency. SAS® data step arrays have a number of unique characteristics that make them especially useful in enhancing your coding productivity. This presentation will provide a useful tutorial on the rationale for arrays and their definition and use.

INTRODUCTION

Most of the row-wise functionality required for data collection, reporting and analysis is handled very efficiently by the various procedures available within SAS. Column-wise processing is a different story. Data may be supplied in forms that require the user to process *across* the columns *within* a single row to achieve the desired result. Column-wise processing typically requires the user to employ the power and flexibility of the SAS data step.

Even within the SAS data step different methods with varying degrees of coding and execution efficiency can be employed to deal with column-wise processing requirements. Data step arrays are unparalleled in their ability to provide efficient, flexible coding techniques to work with data across columns, e.g. a single row of data containing 12 columns of monthly data.

This paper will cover array definition, initialization, use and the efficiencies afforded by their use.

DEFINING ARRAYS

Array definition in SAS is quite different than most other computer languages. In other languages, arrays are typically a temporary container that holds single or multi-dimensional data that has convenient indexing capabilities which may be used to iterate over the data items within, or to directly reference specific items. SAS's _temporary_ array structures are most like the definitions found in other languages, but SAS provides so much more array functionality than found in other languages.

The basic array definition statement is as follows:

```
ARRAY array_name (n) <$>  <length>  array-elements  <(initial-values)>;
```

- array_name — any valid SAS variable name
- (n) — number of array elements (optional in some cases)
- $ — indicates a character array
- length — length of each array element (optional in some cases)
- array-elements — _temporary_ or SAS variable name(s) or variable lists (optional)
- initial-values — array initial values

SAS arrays must contain either numeric or character data, not a mixture of both.

TEMPORARY ARRAY DEFINITION

Temporary arrays provide convenient tabular data stores which persist only while the data step is executing. The "variables" created by the temporary array are automatically dropped from the output dataset. Temporary arrays are often used to act as accumulators, store factors etc.. when the items being referenced are not required to be stored with the data. Temporary arrays are automatically retained across data step iterations, i.e. the element values are not set to missing at the top of the data step like most other data step variables.

```
array factors(12,3) _temporary_ ;
array desc(12)      $15    _temporary_ ;
```

The first statement defines a two-dimension numeric array. The second, a 12 element character array, each element 15 bytes in length.

PDV VARIABLE ARRAY DEFINITION

Non-temporary data step arrays demonstrate the real flexibility and versatility of SAS arrays. These types of arrays can refer to a series of variables already present in the program data vector (PDV) or even create variables while permitting those variables to be referenced via array structures. Some examples will illustrate the possibilities:

Consider an existing dataset containing four columns of quarterly metrics:

VIEWTABLE: Work.Quarterly_data

	cust_id	qtr1	qtr2	qtr3	qtr4
1	1	185	971	400	260
2	2	922	970	543	532
3	3	50	67	820	524
4	4	854	68	958	298
5	5	273	690	977	227
6	6	689	413	559	288
7	7	476	845	635	591

In the data step below, the data step compiler recognizes the columns in the incoming dataset (SET statement) and adds them to the PDV. The array statement references the four quarterly variables and makes them available within an array structure simply called Q. It is not necessary to define the number of elements in the array in this case, SAS will determine the size of the array by the number of array elements defined by the qtr1 – qtr4 variable list.

Note the variable name displayed when the array elements are PUT into the log. The array statement has not created *any additional* variables, but has merely made the pre-existing PDV variables available via a convenient array structure. The array definition can be seen as a *logical* structure overlaying the *physical* variables.

```
set quarterly_data;

array q      qtr1-qtr4;

do _i = 1 to 4;
      put cust_id= @13 q(_i)=;
end;
```

```
cust_id=1   qtr1=185
cust_id=1   qtr2=971
cust_id=1   qtr3=400
cust_id=1   qtr4=260
cust_id=2   qtr1=922
cust_id=2   qtr2=970
cust_id=2   qtr3=543
```

In the following data step, the ARRAY statement will create qtr1, qtr2, qtr3, qtr4 variables in the PDV as per the array definition since they don't already exist.

```
data quarterly_data;
      length cust_id            8;
      array q  8  qtr1-qtr4;
      do cust_id = 1 to 7;
            do _i = 1 to 4;
                  q(_i) = ceil(ranuni(1) * 1000);
            end;
            output;
      end;
      drop _: ;
run;
```

The following statement also defines the variables, but uses the array name to provide the prefix of the PDV variable name and suffixes the name with consecutive numbers. Again, qtr1, qtr2, qtr3, qtr4 will be created at compile time.

```
array qtr(4) 8;
```

The "QTR" array name might look familiar – that's because it's also a built-in SAS function. The compiler takes note of that fact and warns the user that the QTR() function is unavailable in this data step because the array reference will take precedence.

```
366      array qtr(4) 8;
NOTE: The array qtr has the same name as a SAS-supplied or user-defined function.  Parentheses
      following this name are treated as array references and not function references.
```

The first example showed the use of the qtr1 – qtr4 variable list to define the array elements. In the same way, any variable list syntax may be used:

```
array desc    _character_;        * character vars already defined in PDV;

array amts    _numeric_;          * numeric vars already defined in PDV;
```

Arrays may be made up of variables of different lengths. Note the variable list below which specifies that all character variables between flag_x and desc (as defined by the PDV order) be part of the LST array though each character variable has a different length and they are not contiguous.

```
length        flag_x        $10
              amt           8
              name          $20
              dol           4
              desc          $60 ;

array lst     flag_x -character- desc;
```

An instance of a numeric array made up of various numeric variables, i.e. without a common prefix like "qtr":

```
array various       discount refund pre_tax after_tax payable received;
```

By default data step arrays are indexed starting at 1, i.e. the first element in the array above would be referenced by various(1). The array size, i.e. the number of elements, may be derived using the DIM() function. However, it's often advantageous to begin indexing an array with a value other than 1, e.g. in the event we wanted to use the age of school-age children (5-18 years old) directly as an index into an array: An example will make this more clear. Note the use of the various functions which describe the array:

- DIM() which provides the number of array elements
- LBOUND() the index value of the lower array boundary
- HBOUND() the index value of the upper array boundary

```
data _null_;
      array ages (5:18) _temporary_;
      d = dim(ages);
      l = lbound(ages);
      h = hbound(ages);
      put 'Ages array has ' d ' elements, bounded by ' l ' and ' h;

      do until(done);
            set sashelp.class end = done;
            ages(age) + 1;
      end;

      do i = lbound(ages) to hbound(ages);
            put i 2. @7 ages(i);
```

```
        end;
        stop;
run;
```

The log results follow, showing the array element ranges and the number of children in each age range.

```
Ages array has 14  elements, bounded by 5  and 18
 5    .
<snip>
10    .
11    2
12    5
13    3
14    4
15    4
16    1
17    .
18    .
```

Multi-dimensional arrays are defined and dealt with in a very similar manner. Note the extra parameter in the DIM() function to reference to the 2nd dimension. In similar fashion, both LBOUND() and HBOUND() may use the 2nd function parameter to refer to specific dimension as well. Log results follow.

```
data _null_;

        array ages(5:18,2) _temporary_;
        array gender(2) $1 _temporary_ ('M','F');

        d1 = dim(ages);      * number of elements in 1st dimension;
        d2 = dim(ages,2);    * number of elements in 2nd dimension;

        l = lbound(ages);
        h = hbound(ages);
        put 'Ages array has ' d1 ' x ' d2 ' elements, bounded by ' l ' and ' h;

        do until(done);
                set sashelp.class end = done;
                if sex = 'M' then s = 1; else s = 2;
                ages(age,s) + 1;
        end;

        put 'Age' @6 'Male' @12 'Female';
        do i = lbound(ages) to hbound(ages);
                put i @7 ages(i,1) 1.  @15 ages(i,2);
        end;
        stop;
run;

Ages array has 14  by 2  elements, bounded by 5  and 18
Age  Male  Female
<snip>
10    .       .
11    1       1
12    3       2
13    1       2
14    2       2
15    2       2
```

```
16    1       .
17    .       .
18    .       .
```

ARRAY VALUE INITIALIZATION

Arrays may be initialized at compile time with specific values. Since arrays based on PDV variables are really just a method of referencing "normal" variables, the elements in PDV arrays will be set to missing at the top of the data step as per the usual data step rules.

It's helpful to assign initial values to the array at compile time since that saves a step during execution. The first two array initialization statements below accomplish the same thing, i.e. (12*0) is the equivalent of twelve zeroes:

```
data _null_;
      array mths1 (12)    _temporary_  (12*0);
      array mths2 (12)    _temporary_  (0 0 0 0 0 0 0 0 0 0 0 0);
      array mth_qtr (12)  _temporary_  (3*1 3*2 3*3 3*4);
      do i = 1 to 12;
            put mths1(i)= @14 mths2(i)= @27 mth_qtr(i)=;
      end;
run;

mths1[1]=0   mths2[1]=0   mth_qtr[1]=1
mths1[2]=0   mths2[2]=0   mth_qtr[2]=1
mths1[3]=0   mths2[3]=0   mth_qtr[3]=1
mths1[4]=0   mths2[4]=0   mth_qtr[4]=2
<snip>
mths1[11]=0  mths2[11]=0  mth_qtr[11]=4
mths1[12]=0  mths2[12]=0  mth_qtr[12]=4
```

EFFICIENCIES

Using arrays in the data step will result in coding efficiency, more maintainable code and at times, data-driven code. Data steps that make good candidates for array processing are those that contain "wallpaper code", i.e. a repeating pattern of code where multiple values of the same type are processed, e.g. month1 to month12 values, or calculations involving factors that can be indexed by another data value (e.g. age).

In addition, the SAS summary functions such as SUM() are able to receive an entire array as an input parameter, thus simplifying the code.

STRIPPING WALLPAPER

The most common use for arrays is the ability to iterate over a series of similar variables, performing the same operation on each, without having to code each variable manually.

```
data monthly_sales;
      set sales;
      if month1    = . then month1 = 0; else month1=round(month1,.1);
      if month2    = . then month2 = 0; else month2=round(month2,.1);
      ....
      if month11   = . then month11 = 0; else month11=round(month11,.1);
      if month12   = . then month12 = 0; else month12=round(month12,.1);
run;
```

The use of an array reduces the lines of code from 12 to 5:

```
data monthly_sales;
      set sales;
      array m month1-month12;
      do _i = 1 to dim(m);
            if m(_i) = . then m(_i) = 0; else m(_i) = round(m(_i),.1);
      end;
      drop _: ;
```

```
run;
```

It may be necessary to assign a different factor to a metric depending on a third data item. e.g. dosage variance by age. Rather than coding a series of IF / THEN / ELSE or SELECT / WHEN statements, an array can be employed to lookup the dosage values.

```
data dosages;
      set sashelp.class;
      array dosages (5:18) _temporary_ ( .5 .5 .6 .7 1 1.2 1.3 1.5
                                         1.6 1.8 2.1 2.3 2.7 3.0);
      dosage = dosages(age);
run;

proc print data = dosages noobs;
      var name age dosage;
run;
```

```
Name       Age    dosage

Alfred      14      1.8
Alice       13      1.6
Barbara     13      1.6
Carol       14      1.8
Henry       14      1.8
James       12      1.5
Jane        12      1.5
Janet       15      2.1
Jeffrey     13      1.6
John        12      1.5
Joyce       11      1.3
Judy        14      1.8
Louise      12      1.5
Mary        15      2.1
Philip      16      2.3
Robert      12      1.5
Ronald      15      2.1
Thomas      11      1.3
William     15      2.1
```

FUNCTIONS WITH ARRAYS

Many SAS functions can process an entire array with a very simple coding reference, somewhat akin to how variable lists can be processed by these same functions. In the example below in the second data step, the SUM() function is coded two different ways and there's no real advantage to using an array.

```
data sales;
      do cust_id = 1 to 10;
            array mth(12) 8;    * defines mth1 - mth12 to PDV ;
            do _i = 1 to 12;
                  mth(_i) = ceil(ranuni(1) * 1000);
            end;
            output;
      end;
      drop _: ;
run;

data monthly_sales;
      set sales;
      array m mth1-mth12;
      annual_sum_array = sum(of m(*) );
```

```
        annual_sum_list  = sum(of mth: );
        drop mth: ;
run;
```

Results of the array and variable list summation are identical:

VIEWTABLE: Work.Monthly_sales

	cust_id	annual_sum_array	annual_sum_list
1	1	6244	6244
2	2	6294	6294
3	3	7494	7494
4	4	5643	5643
5	5	4777	4777
6	6	5811	5811
7	7	4114	4114
8	8	6106	6106
9	9	5790	5790
10	10	6402	6402

But, if the monthly sales variables were named JAN, FEB, MAR etc… and mixed through the dataset among other numeric variables, there would be no way to reference them via a variable list and arrays would be the only method of referencing them efficiently. e.g. `array m jan feb mar … dec;`

SAS also provides a call routine to sort array contents in ascending order, SORTC and SORTN for character and numeric arrays respectively.

```
data _null_;
   array name(8) $10
      ('George' 'nancy' 'Susan' 'george'
       'James' 'Robert' 'Rob' 'Fred');
   call sortc(of name(*));
   put +3 name(*);
run;

Fred George James Rob Robert Susan george nancy
```

SPECIAL ARRAY FUNCTIONS

It's all well and good to define an array that contains a number of character or numeric variables in the PDV. But, sometimes when the array is processed, it's very helpful to be able to identify the characteristics of the specific variables underlying each array element. There's a number of functions available, some listed below, that surface the particulars of the variable underlying the array reference. See SAS Online Documentation for a complete list:

VNAME()	variable name
VLABEL()	label of the variable
VFORMAT()	format applied to variable
VLENGTH()	defined variable length

The real-world example below uses arrays and the VFORMAT function to create a fixed length text file.

Define the skeleton dataset containing the variables and their formats:

```
data dialer_skeleton;
      format num          $12.   name         $20.
             expiry       $10.   msg          $15.
             rep          $3.    ;

      array txtfile _character_;
      call symputx('dim',dim(txtfile));       * capture array length;
      stop;
run;
```

Define the test data:

```
data dialer_input;
      length num    $10    name   $20
             msg    $15    rep    $2;

      num = '9052941234'; name = 'George Smith';     expiry = '31Oct2012'd;
      msg = '1st call';   rep = '01';  output;
      num = '5196471212'; name = 'Susan Jones';      expiry = '01Nov2012'd;
      msg = 'Second call';rep = '01';  output;
      num = '4162235678'; name = 'Sam Brown';        expiry = '02Nov2012'd;
      msg = 'New customer';rep = '2';  output;
      num = '6137219988'; name = 'Brian Tait';expiry = '29Oct2012'd;
      msg = 'Attriter';   rep = '2';   output;
run;
```

Create output text data. Note the use of the VFORMAT function to retrieve the formatted length to ensure each data item will be written with the correct length to line up properly in the fixed-width text output.

```
%macro txt;
      data dialer ( drop = _: );
            if 0 then set dialer_skeleton;    * set variable lengths/formats;

            array txtfile _character_;        * make available in array;

            set dialer_input ( rename = ( expiry = _exp ));

            expiry = put(_exp,yymmddd10.);
            length record $100;
            record = putc(txtfile(1),vformat(txtfile(1)))
                        %do i = 2 %to &dim; * loop limit from skeleton step;
                              || putc(txtfile(&i),vformat(txtfile(&i)))
                        %end;
                        ;
            put record;
      run;
%mend txt;

%txt
```

Results:

```
9052941234  George Smith        2012-10-311st call       01
5196471212  Susan Jones         2012-11-01Second call    01
4162235678  Sam Brown           2012-11-02New customer   2
6137219988  Brian Tait          2012-10-29Attriter       2
```

The flexibility of this method comes into play when the text output requirements change. The only code that need be modified is the dialer_skeleton data step. Since the data step that actually creates the text output has no hard-coded variables or lengths, any changes will be picked up automatically. Less maintenance = less errors.

CONCLUSION

SAS data step arrays are simple to define and provide many opportunities for coding efficiency. Common *physical* data characteristics may be overlaid with *logical* array definitions which permit the use of looping constructs and functions to write efficient, flexible and maintainable code. Arrays have other uses as well which have been covered by other fine SAS user group papers, e.g. data step transpose. See www.lexjansen.com and search for "array" for additional reading.

REFERENCES

Steve First and Teresa Schudrowitz, "Arrays Made Easy: An Introduction to Arrays and Array Processing" http://www2.sas.com/proceedings/sugi30/242-30.pdf

SAS Institute 2009, "SAS 9.4 Language Reference: Dictionary", http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#p1i7zlvukj8arhn1ihivqz9mot9m.htm

CONTACT INFORMATION

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

Harry Droogendyk
conf@stratia.ca
Stratia Consulting Inc.
www.stratia.ca