PharmaSUG China 2018 – Paper CC-12

Advance Array Processing Techniques: Ingenious uses of Array for SAS

Gladys Vanessa Espinas, PPD, Manila, Philippines
Eduard Joseph Siquioco, PPD, Manila, Philippines

ABSTRACT

An array is practically used to group variables for processing within a single SAS DATA step. A basic application of an array is to easily perform repetitive tasks on a set of similar columns. Another task that can be simplified by using an array is finding a specific value in a series of variables. There are several other ingenious uses of an array yet programmers tend to be unaware of how convenient and useful it is in SAS DATA step programming.

This paper will explore advanced array techniques in programming derived datasets such as complex date calculations, in-between date merging, and robust conditional processing. This paper is aimed to enhance the data manipulation techniques, and to have a more concise and efficient SAS codes of programmers.

INTRODUCTION

A SAS array groups together variables for processing within a single SAS DATA step. Using arrays can shorten program codes and simplify repetitive conditional processing statements. Aside from easily performing similar tasks on a set of columns, there is more to know about ingenious and convenient applications of SAS arrays that programmers can benefit from.

BASIC CONCEPTS

Defining Arrays

The syntax of defining a SAS array is executed by using the ARRAY statement.

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

Component	Description
array-name	a valid SAS name that will be used to identify the group of variables
n	a subscript that refers to the number of elements or variables in the array that should be enclosed in either parentheses ( ), braces { }, or brackets [ ]
$	a component added to the statement to specify that the elements within the array are of character type
length	a common length of the elements in the array
array-elements	a list of the variables to be part of the array that must be all the same type, either all numeric or all character
initial-value-list	a list of initial values for each of the array elements

Table 1. Components of an ARRAY statement

The following SAS codes are simple examples of defining arrays:

```
array charsamp {5} $ 10 varc1 varc2 varc3 varc4 varc5;

array numsamp (*) var1 var2 var3;

array initsamp [2] $ init1 init2 ('A', 'B');
```

In the *numsamp* array example, the asterisk subscript {*} specifies that SAS is to automatically determine and count the number of elements within the array.

To refer to an array defined within the same SAS DATA step, an array reference statement is used.

```
array-name {subscript}
```

The array-name refers to the name of an array previously defined with an ARRAY statement. The subscript component represents the value of an element's position defined within the array. In the *charsamp* array example, the array reference for *varc2* variable will be specified as follows:

```
charsamp {2}
```

Using Arrays

A SAS array is commonly used or processed in iterative DO loops with IF-THEN-ELSE conditions. To illustrate, a simple iterative DO loop such as follows can be executed to perform the same action on all elements of the *numsamp* array example:

```
do i=1 to 5;
 if numsamp[i] =. then numsamp[i] = 0;
end;
```

For a more dynamic DO loop in ARRAY processing, the DIM function can be utilized.

```
dim (array-name)
```

The DIM function returns the total count of the number of elements of the specified array. Using DIM with *numsamp* array will return a value of 5 and can be used as the upper bound value of the previous DO loop expression.

```
do i=1 to dim(numsamp);
 if numsamp[i] =. then numsamp[i] = 0;
end;
```

DIM is often used in place of the static numeric upper bound value of the DO loop to avoid altering the value each time the number of array elements changes. Using DIM in such cases results to a robust DO loop in ARRAY processing.

INGENIOUS USES OF ARRAYS

This section will present some practical applications of SAS arrays with additional and relevant advanced programming techniques in ARRAY processing.

FORMATTING OF MULTIPLE VARIABLES

Raw data sources usually consist of multiple variables with values that require similar decoding for data processing. Using arrays in this situation is beneficial by easing the repetitive programming codes. To demonstrate, consider *sample_sae1* dataset below as the raw data with several variables containing Yes or No values depicted as 1 or 0, respectively.

VIEWTABLE: Work.Sample_sae1

	SUBJECT	_AESDTH	_AESLIFE	_AESHOSP	_AESDISAB	_AESCONG	_AESMIE
1	1001	0	1	1	0	0	0
2	1002	1	1	1	0	0	0
3	1003		0	1	0	0	

VIEWTABLE: Work.Sample_sae2

	SUBJECT	AESDTH	AESLIFE	AESHOSP	AESDISAB	AESCONG	AESMIE
1	1001	N	Y	Y	N	N	N
2	1002	Y	Y	Y	N	N	N
3	1003		N	Y	N	N	

Display 1. *Sample_sae1* and *Sample_sae2* datasets

The common program to decode 1 to 'Y' and 0 to 'N' as in the *sample_sae2* dataset will include several and similar coding statements. With ARRAY and DO loop, lines can be reduced like in the following code:

```
data sample_sae2;
 set sample_sae1;
  array saen {*} _aesdth _aeslife _aeshosp _aesdisab _aescong _aesmie;
  array saec {*} $1 aesdth aeslife aeshosp aesdisab aescong aesmie;
  do i=1 to dim(saec);
   if saen{i} = 1 then saec{i} = 'Y';
   else if saen{i} = 0 then saec{i} = 'N';
  end;
run;
```

The iterative DO loop decodes all the input variables, defined in the *saen* array, to the corresponding formatted output variables, defined within *saec* array, with only one set of IF-THEN-ELSE statements.

It is important to note that the order of the variables should be listed in each array correspondingly when defining several associated arrays within a single DATA step.

Custom Formats

A custom format *YN* can be created with a FORMAT procedure to further ease the DO loop expression.

```
proc format;
 value YN
   1 = 'Y'
   0 = 'N';
run;
```

Applying the custom format *YN* with PUT function will result to having the IF-THEN-ELSE statements within the DO loop reduced to one IF-THEN statement.

```
do i=1 to dim(saec);
 if ~missing(saen{i}) then saec{i} = put(saen{i},yn.);
end;
```

The FORMAT procedure should be specified prior to the DATA step where the user-created format will be used.

SEARCHING MULTIPLE VARIABLES

Another task that can be simplified by using an array is finding a specific value in a series of variables. For this application, consider the *sample* dataset with flag variables, *flag1* to *flag3*, with 'Y' or 'N' values. The desired output *sampleproc1* dataset includes the following derived variables:

1. *miss_flag*: will contain a value of 'Y' if any of *flag1* to *flag3* variables is a missing value

2. *n_flag*: will contain a value of 'Y' when *flag1* to *flag3* values is 'N'

VIEWTABLE: Work.Sample

	SUBJECT	FLAG1	FLAG2	FLAG3
1	AAA	Y	Y	Y
2	BBB	Y	N	
3	CCC	N	N	N

VIEWTABLE: Work.Sampleproc1

	SUBJECT	FLAG1	FLAG2	FLAG3	MISS_FLAG	N_FLAG
1	AAA	Y	Y	Y		
2	BBB	Y	N		Y	Y
3	CCC	N	N	N		Y

Display 2. *Sample* and *Sampleproc1* datasets

To identify if there are missing or 'N' values among the flag variables, an array can be created to group *flag1* to flag3.

```
data sampleproc1;
 set sample;
  array flag {3} $ flag1 flag2 flag3;
   if ("") in flag then miss_flag="Y";
   if ("N") in flag then n_flag="Y";
run;
```

Short ARRAY Statements

If the names of the variables in an array are prefixed similarly, the array can be defined without listing each element. Using the *flag* array as a reference example, the following are short variant ways of defining an ARRAY:

```
array flag {3} $ flag1-flag3;

array flag {*} $ flag:;

array flag {3} $;
```

When all numeric or all character variables of a dataset are to be grouped in an array, the SAS variable lists _NUMERIC_ or _CHARACTER_ can be used to reference the element list, respectively. From an earlier example, the *saen* array can be defined similarly with the variable list _NUMERIC_.

```
array saen {*} _numeric_;
```

The SAS variable list _ALL_ can also be used to reference the array element list if all variables of the dataset are to be elements of the array and all are of the same data type.

Dynamic Element List using Macro Variables

In cases where the array elements do not have the same prefix but can be identified with similar text strings, an SQL procedure such as below can be used to create a dynamic list of the array variables in a macro variable.

```
proc sql noprint;
 select distinct NAME into :NEWFLAGS separated by " "
 from dictionary.columns
 where libname = "WORK" and memname = "SAMPLEPROC1" and
       upcase(NAME) contains "_FLAG";
quit;
```

The resulting macro variable *newflags* can now be used in the ARRAY statement as the element list.

```
    array newfl {*} &newflags.;
```

CONCATENATION OF VALUES

This section is similar to the previous application with additional uses of array in concatenation of values.

In the following *sample_recon2* dataset, a *miss_n_flag* variable is created to summarize which mismatch variables have missing or 'N' values.

VIEWTABLE: Work.Sample_recon1

	Subject	Date mismatch	Time mismatch	Visit mismatch
1	1001	Y	Y	Y
2	1002	Y	N	
3	1003	Y	N	Y

VIEWTABLE: Work.Sample_recon2

	Subject	Date mismatch	Time mismatch	Visit mismatch	MISS_N_FLAG
1	1001	Y	Y	Y	
2	1002	Y	N		Time mismatch, Visit mismatch
3	1003	Y	N	Y	Time mismatch

Display 3. *Sample_recon1* and *Sample_recon2* datasets

Temporary ARRAY

A temporary array is useful for holding constant values used in calculations. In defining a temporary array, the ARRAY statement is used with the SAS key word _TEMPORARY_ as the array-name and no corresponding variables will be listed as the array elements. Temporary data elements are automatically created in the program data vector and are set with the initial values indicated. Note that the asterisk subscript {*} cannot be used with temporary arrays.

```
data sample_recon2;
 set sample_recon1;
 length miss_n_flag $200;
  array flag{3} $;
  array flagd{3} $100 _temporary_ ('Date mismatch' 'Time mismatch' 'Visit mismatch');
  do i=1 to dim(flag);
    if flag{i} in ("" "N") then do;
     if missing(miss_n_flag) then miss_n_flag = flagd{i};
     else miss_n_flag = strip(miss_n_flag)||', '||flagd{i};
    end;
  end;
run;
```

The *miss_n_flag* variable is created using a temporary array to hold the text values of the variable labels needed for concatenation.

APPLYING FUNCTIONS

As seen in the previous applications, array reference statements can be used as an argument in SAS functions. Similarly, an array refence can also be utilized as a macro variable value or as a parameter value in a system or user-defined macro functions in SAS.

VIEWTABLE: Work.Sample_macro1

	SUBJECT	WINDOWST	WINDOWEN	DATE1	DATE2	DATE3
1	1001	17MAR2017	05SEP2017	05SEP2017	01AUG2017	10FEB2017
2	1002	01APR2018	23MAY2018	02APR2018	17MAR2018	14AUG2018
3	1003	30DEC2016	12DEC2017	06SEP2017	26AUG2017	04NOV2017

VIEWTABLE: Work.Sample_macro2

	SUBJECT	WINDOWST	WINDOWEN	DATE1	DATE2	DATE3	MAXDT
1	1001	17MAR2017	05SEP2017	05SEP2017	01AUG2017		05SEP2017
2	1002	01APR2018	23MAY2018	02APR2018			02APR2018
3	1003	30DEC2016	12DEC2017	06SEP2017	26AUG2017	04NOV2017	04NOV2017

Display 4. *Sample_macro1* and *Sample_macro2* datasets

```
%macro within_window (date=);

  if not ((.<windowst<=&date<=windowen) or
          (.<windowst<=&date and missing(windowen))) then &date = .;

%mend within_window;

data sample_macro2;
 set sample_macro1;
  array date{3};
  do i=1 to dim(date);
    %within_window(date=date{i});
  end;
  maxdt=max(of date{*});
run;
```

In this example, the array reference for *date* array is used as the parameter value for the user-defined *within_window* macro function. The *date* array reference with asterisk subscript {*} is also used as the argument in MAX function to derive the maximum date variable, *maxdt*. The asterisk subscript refers to all elements of the array. In this case, the argument list of the MAX function consists of the all variables defined in the *date* array.

MATCH CORRESPONDING VALUES

Arrays can also be utilized in instances where a derivation is dependent on which values match from a group of variables. For example, values of two certain columns, censor (*cnsr*) and event description (*evntdesc*), will be derived based on which among the 3 date variables is equivalent to the maximum date (*maxdt*).

1. If maximum date is equal to withdrew consent date *(date1*) then censor value is 1 and event description value is 'Withdrew consent'
2. If maximum date is equal to lost to follow-up date *(date2*) then censor value is 2 and event description value is 'Lost to follow-up'
3. If maximum date is equal to last contact date *(date3*) then censor value is 3 and event description value is 'Ongoing'

VIEWTABLE: Work.Sample_os2

	SUBJECT	Withdrew consent date	Lost to follow-up date	Last contact date	MAXDT	CNSR	EVNTDESC
1	1001	05SEP2017	01AUG2017		05SEP2017	1	Withdrew consent
2	1002	02APR2018			02APR2018	1	Withdrew consent
3	1003	06SEP2017	26AUG2017	04NOV2017	04NOV2017	3	Ongoing

Display 5. *Sample_os2* dataset

```
data sample_os2;
 set sample_os2;
  array date {3} date1 date2 date3;
  array _cnsr {3} _temporary_ (1 2 3);
  array desc {3} $100 _temporary_ ('Withdrew consent' 'Lost to follow-up' 'Ongoing');
  do j=1 to dim(date);
   if maxdt = date{j} then do;
      cnsr = _cnsr{j};
      evntdesc = desc{j};
   end;
  end;
run;
```

As a case in point, arrays were used in deriving the censor and event description. The three date variables were grouped into one array and two temporary arrays were defined to hold the corresponding values for censor and description.

CARRY OVER VALUES

Array reference of the same array is not limited to one reference per statement or line code. For this reason, an array is useful in carrying over values from multiple variables to another.

To illustrate, each missing visit score variable in *sample_visit1* dataset below should be filled in with the last non-missing value of the previous visits as in *sample_visit2* dataset.

VIEWTABLE: Work.Sample_score1

	SUBJECT	VISIT_SCR1	VISIT_SCR2	VISIT_SCR3	VISIT_SCR4
1	1001	3	3	4	5
2	1002	4	3	.	.
3	1003	5	.	.	.
4	1004	2	2	3	.
5	1005	.	1	2	.

VIEWTABLE: Work.Sample_score2

	SUBJECT	VISIT_SCR1	VISIT_SCR2	VISIT_SCR3	VISIT_SCR4
1	1001	3	3	4	5
2	1002	4	3	3	3
3	1003	5	5	5	5
4	1004	2	2	3	3
5	1005	.	1	2	2

Display 6. *Sample_score1* and *Sample_score2* datasets

Using an array in this case can be done and should require an array reference to both the current and the previous visit score variables.

```
data sample_score2;
 set sample_score1;
  array visit{*} visit_:;
  do i=2 to dim(visit);
    if missing(visit{i}) then visit{i} = visit{i-1};
  end;
run;
```

The array reference *visit {i}* refers to the current visit score while the *visit {i-1}* refers to the previous visit score variable. The DO loop lower bound value of *i* was initialized to 2 as *VISIT_SCR1* has no previous visit score. The array reference for the previous visit score will not yield a result with a subscript equal to 0.

COMPLEX DATE DERIVATIONS

Some derivations concerning date variables can be complicated and tricky. Unknown to most programmers, arrays are useful in dealing with date-related calculations. To show how arrays are of use in situations with date derivations, consider the following mock clinical study data:

VIEWTABLE: Work.Sample

	SUBJECT	CMSTDT	CMENDT	SEQ
1	AAA	01JAN2018	10JAN2018	1
2	AAA	12JAN2018	20JAN2018	2
3	AAA	25JAN2018	30JAN2018	3

VIEWTABLE: Work.Visits

	SUBJECT	AVISIT	VSTDT
1	AAA	VISIT A	10JAN2018
2	AAA	VISIT B	15JAN2018
3	AAA	VISIT C	31JAN2018

Display 7. Mock clinical study data, *Sample* and *Visit* datasets

During and prior to subject *AAA* visits, the subject was given concomitant medications and the medication dosing periods, start (*cmstdt)* and end (*cmendt)* dates, are recorded in the *sample* dataset. The visit information of subject *AAA* is contained in the *visits* dataset. The intent with these data is to calculate the cumulative dosing duration at each visit. For visualization, derivation of the dosing duration for each visit is illustrated by the following timeline:



VISIT A cumulative dosing duration: number of days between the Blue line = **10**
VISIT B cumulative dosing duration: number of days between Blue + Red lines = **14**
VISIT C cumulative dosing duration: number of days between Blue + Red + Yellow + Purple lines = **25**

Figure 1. Illustrated timeline of the dosing durations

To utilize arrays to derive the cumulative dosing duration for this example, the input datasets should be restructured and merged for ARRAY processing compatibility.

VIEWTABLE: Work.Sample

	SUBJECT	CMSTDT	CMENDT	SEQ
1	AAA	01JAN2018	10JAN2018	1
2	AAA	12JAN2018	20JAN2018	2
3	AAA	25JAN2018	30JAN2018	3

Transpose the *sample* dataset into a wide structure using SEQ as ID variable

VIEWTABLE: Work.Sample_wide

	SUBJECT	START_1	START_2	START_3	END_1	END_2	END_3
1	AAA	01JAN2018	12JAN2018	25JAN2018	10JAN2018	20JAN2018	30JAN2018

Merge the *sample_wide* dataset to *visits* dataset by SUBJECT to get all the information in one dataset

VIEWTABLE: Work.Sample_array

	SUBJECT	START_1	START_2	START_3	END_1	END_2	END_3	AVISIT	VSTDT
1	AAA	01JAN2018	12JAN2018	25JAN2018	10JAN2018	20JAN2018	30JAN2018	VISIT A	10JAN2018
2	AAA	01JAN2018	12JAN2018	25JAN2018	10JAN2018	20JAN2018	30JAN2018	VISIT B	15JAN2018
3	AAA	01JAN2018	12JAN2018	25JAN2018	10JAN2018	20JAN2018	30JAN2018	VISIT C	31JAN2018

Display 8. Transformation of the input datasets for ARRAY processing

```
data sample_process;
 set sample_array;
 retain value 0; *Use a retained variable to sum up correctly;
  if first.vstdt then value = 0; *Reset retained variable in next record;
  *Define the arrays for start and end dates;
  array start{*} start:;
  array end{*} end:;
  *Calculate the dosing duration;
  do i =1 to dim(start);
   *Add duration as is when VSTDT >= ENDDATE;
    if end[i] <= vstdt then value = value + (end[i]- start[i]+1);
   *Change end date to VSTDT when in between doses;
    else if end[i] > vstdt > start[i] then value = value + (vstdt - start[i]+1);
  end;
run;
```

VIEWTABLE: Work.Sample_process

	VALUE	SUBJECT	AVISIT	VSTDT
1	10	AAA	VISIT A	10JAN2018
2	14	AAA	VISIT B	15JAN2018
3	25	AAA	VISIT C	31JAN2018

Display 9. *Sample_process* dataset

CONCLUSION

This paper explores data manipulation techniques and some ingenious uses of arrays in SAS. The convenient use of arrays in DATA step programming can provide a more concise and efficient program codes.

ACKNOWLEDGMENTS

The authors would like to thank Thea Valerio for her contributions in the development of this paper.

RECOMMENDED READING

- Arrays Made Easy: An Introduction to Arrays and Array Processing:
  http://www2.sas.com/proceedings/sugi30/242-30.pdf
- Effectively Utilizing Loops and Arrays in the DATA Step:
  https://www.pharmasug.org/proceedings/2013/HT/PharmaSUG-2013-HT04.pdf

CONTACT INFORMATION

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

Name: Gladys Vanessa Espinas
Enterprise: PPD Inc.
Address: Manila, Philippines
E-mail: GladysVanessa.Espinas@ppdi.com

Name: Eduard Joseph Siquioco
Enterprise: PPD Inc.
Address: Manila, Philippines
E-mail: EduardJoseph.Siquioco@ppdi.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.