Happiness is Using Arrays to Make Your Job Easier

Greg M. Woolridge, Searle, Skokie, IL
Winnie Lau, ASG, Inc, Vernon Hills, IL

Abstract

Arrays and array processing are very powerful tools available in the SAS® system which can be used to make programming easier. However; many people don't use them because they are thought to be too confusing or difficult to learn and use. This paper will present a brief tutorial on defining arrays and their properties. We will also demonstrate some real-life examples of how they can be used to make writing programs easier.

Introduction

There are 2 types of arrays available for use in the SAS system, implicit and explicit. The most noticeable difference between the 2 types is that explicit arrays have a subscript in their definition which defines the number and arrangement of elements in the array. The way in the elements are referenced is also different.

SAS supports implicit arrays so that code written under earlier versions of SAS, before explicit arrays were available, is compatible with newer versions of SAS. Explicit arrays are recommended for use in all new coding and will be the only ones discussed in this paper. All references to arrays are considered to be explicit arrays unless otherwise stated.

Description

Arrays are data structures which can be used in a DATA step to make coding simpler. Arrays exist only for the duration of a DATA step and are not stored as part of any data set. The elements of an array can be stored as part of a data step if they are not temporary variables and are not dropped as the result of a DROP or KEEP option or statement.

Arrays can have as their elements numeric or character variables, but the variables of an array must be all of the same type. Temporary variables can also be used as the elements of an array, but they have some special properties which will be discussed below.

The ARRAY STATEMENT

Arrays are defined using an ARRAY statement. This statement is only valid as part of a DATA step. The syntax of the ARRAY statement is:

ARRAY array-name{subscript} $ length *elements (initial values)*;

The **array name** parameter is required and must follow the standard conventions for a SAS variable name (i.e., no longer than 8 characters or numbers and begin with a character). You may not use as the array name the name of any variable defined in the program data vector for the DATA step. It is also a good idea not to name an array with the name of a SAS function, although SAS will accept those names as valid.

The **subscript** parameter defines the number and arrangement of elements in an array. The subscript must be enclosed in braces ({ }), brackets ([ ]) or parentheses (( )) and is usually a number or range of numbers. Data set variables may not be used as the subscript, but macro variables are acceptable.

One-dimensional arrays are the most common type of array used and have a single number as the subscript. Multidimensional arrays may be defined by using numbers separated by commas to define the number of elements in each dimension as the subscript. As multidimensional arrays are a more advanced use of arrays, they will not be discussed in this paper.

If the number of elements is not known at the time the code is written, or may vary, an asterisk (*) may be used. SAS will calculate the subscript based on the number of elements defined.

The individual elements of an array are referenced by using the format **array-name(subscript)** in code where subscript is set to the element number you wish to reference. Subscript may be a number, a numeric variable or a macro variable which returns a number. The elements of the array are assigned numbers which correspond to their position in the element parameter discussed below.

The **dollar sign ($)** parameter indicates that the elements in the array are character. This is optional if the elements have been previously defined as character, and should not be used if the elements are numeric.

The **length** parameter defines the length of each element in the array that has not been previously assigned a length. If previously defined elements are used, this parameter has no effect on the variables. Variables of different length may be used in the same array. If this parameter is omitted and new variables are defined, they are given the SAS default length (200 for character and 8 for numeric).

The **elements** are the variables which make up the array. They must be all character or all numeric within a single array. You can use either data set variables or temporary variables. The variables need not be defined prior to the array statement. If a variable name not previously defined is used as an array element, SAS will define the variable using the length parameter, if present. The variable will be of the same type of the other, predefined, elements. If all elements are to be defined within the array statement, they will be character if the dollar sign parameter is used, or numeric otherwise.

There are 4 special keywords available for defining the elements of an array. **_NUMERIC_** will include all numeric variables defined in the data step as elements. **_CHARACTER_** will include all character variables defined in the data step as elements. **_ALL_** will include all variables defined in the data step as elements. In order to use this keyword, all variables must be of the same type.

The _**TEMPORARY**_ keyword is used to define temporary variables. They can be numeric or character and behave like data set variables with the following exceptions:

- They do not have names. They must be referenced by using the array name and dimension.
- They exist only for the duration of the data step and cannot be included as part of any output data set.
- You can not use the asterisk subscript. You must explicitly define the dimensions of the array.
- Their values are always retained when beginning the next iteration of the data step.

The **(initial values)** parameter is used to give the elements of an array an initial value. Initial values can be assigned to temporary variables as well as data set variables. Parentheses are required to specify values rather than elements. The initial values are matched with the elements by position. If there are more elements than initial values, the remaining elements are assigned a missing value. The initial values are retained until a new value is assigned to the element.

Using Arrays

Now that we have discussed how arrays are defined and some of their properties, let's look at a few real life examples of how arrays can making writing code easier.

Example 1: Change the value of a series of variables.

Suppose that we have a number of variables we want to change the value of. This could be a situation where you want to change the value of missing to 0 in a series of variables. We can use multiple IF statements for processing:

```
DATA PERCENT; SET PERCENT;
      IF C1=. THEN C1=0;
      IF C2=. THEN C2=0;
      IF C3=. THEN C3=0;
      IF C4=. THEN C4=0;
      IF C5=. THEN C5=0;
      IF C6=. THEN C6=0;
RUN;
```

This works fine and is not too cumbersome if the number of variables is small. However; if you have a large number of variables to deal with, writing all those IF statements can become tedious. You also increase the chance of making a mistake in one or more of your statements. Instead of using multiple IF statements to change the value of variables, we can use an array to make the processing of these variables easier.

Our example data set contains the following variables and corresponding values:

C1	***C2***	***C3***	***C4***	***C5***	***C6***
.	*836*	*113*	*.*	*98*	*133*
773	*.*	*.*	*3*	*318*	*.*

You can make use of arrays in the following manner to make writing your code to change the missing values to 0 easier:

```
DATA PERCENT; SET PERCENT;
      ARRAY CNT {6} C1 C2 C3 C4 C5 C6;
      DO C=1 TO 6;
            IF CNT(C)=. THEN CNT(C)=0;
      END;
RUN;
```

In this example, the array statement sets up the array and assigns as its elements the variables C1-C6. The DO loop is then used to process each element of the array. The IF statement inside the DO loop shows how the elements of the array are referenced by using CNT(C). CNT is the name of the array, and the index variable C is used inside parentheses to indicate which element of the array is to be referenced. As C increases from 1 to 6 each element is processed. The resulting data set would be:

C1	***C2***	***C3***	***C4***	***C5***	***C6***
0	*836*	*113*	*0*	*98*	*133*
773	*0*	*0*	*3*	*318*	*0*

You have taken 6 statements using IF processing and reduced it to 4 using an array. While the savings in code is not great in this example, if you had 20 variables to process you would need 20 statements using IF processing, but still only 4 statements using arrays, a savings worth considering.

Example 2: Using arrays to gather information from multiple observations into one observation

In order to calculate the time from dosing to a blood draw, we need to compare the blood draw time (BLDTM) to the various dosing times. In our shop, the dosing records are structured as one observation per dosing time, as shown below.

PT	***DOSEDT***	***DOSESTM***
101	*13MAR98*	*0800*
101	*13MAR98*	*1602*
101	*14MAR98*	*0001*
101	*14MAR98*	*0803*

It is difficult to compare the blood drawn time (in this case 13MAR98:10:00) to the dosing times when all the dosing dates and times are in separate observations. What we need to do is get all of the dosing information on one observation so we can easily compare to the blood draw date and time. This can be done using the TRANSPOSE procedure, but this procedure can be confusing and difficult to use. An easier way to do this is to use arrays.

The first step is to create a series of variables called STIME1-STIME4 which contain all of the dosing dates and times. These can be created in a single observation using this code:

```
DATA DOSE;
      ARRAY STIME {4} STIME1-STIME4;
      DO C=1 TO 4;
            SET DOSE;
            BY PT DOSEDT;
            SDT=DHMS(DOSEDT,INT(DOSESTM/100),
                        MOD(DOSESTM,100),0);
            STIME(C)=SDT;
            IF LAST.PT THEN RETURN;
            KEEP PT STIME1-STIME4;
            FORMAT STIME1-STIME4 DATETIME16.;
      END;
RUN;
```

In this example the array STIME is defined as containing the variables STIME1-STIME4. You should note that the data set variables are named with a number at the end so that STIME without a number is an accepted array name. The SET statement inside the DO loop causes an observation to be read from the input data set with each iteration of the loop. The dosing date and time are combined to create values for each dose which are placed into the STIME1-STIME4 variables. The statement "IF LAST.PT THEN RETURN" triggers an observation to be written to the output data set. The resulting SAS data set will look like:

PT	***STIME1***	***STIME2***	***STIME3***	***STIME4***
101	*13MAR98: 08:00:00*	*13MAR98: 16:02:00*	*14MAR98: 00:01:00*	*14MAR98: 08:03:00*

Next we can calculate the time relative to dosing time by comparing the blood draw time to each of the dosing times:

```
      DO C=1 TO 4 WHILE(STIME(C)^=.);
            IF STIME(C)<=BLDTM<STIME(C+1) THEN
                RELTIME=BLDTM-STIME(C);
      END;
```

This DO loop is placed in a data step which reads the data set created in the previous step. The loop steps through the array elements until it finds the 2 dosing times that the blood draw time (BLDTM) falls between. When these are found, the relative time (RELTIME) is calculated. In this example the RELTIME is calculated to be 2.

Example 3: Stack different variables together for printing.

Many times we have a need to print the values of multiple variables in a single column on a report. For example, when generating Premature Termination listings, we print the Reason for Termination (contained in variables REAS1 and REAS2) and any Comments (contained in variables COM1-COM3) in a single column. We can use IF statements in a DATA _NULL_ to accomplish this:

```
      IF REAS1^='' THEN PUT @80 'R:' @82 REAS1;
      IF REAS2^='' THEN PUT @82 REAS2;
      IF COM1^='' THEN PUT @80 'C:' @82 COM1;
      IF COM2^='' THEN PUT @82 COM2;
      IF COM3^='' THEN PUT @82 COM3;
```

This method works fine if the number of variables used is fixed. When the number of variables can vary, due to a need to accommodate text of varying length, this method has a large potential for error. Each time the number of variables changes, the programmer must change the code in multiple places, and each time the code is changed an error can occur. Instead, we can stack the Reasons for Termination and Comments into an array in a DATA step before DATA _NULL_.

Consider a data set with the following values:

REAS1	***REAS2***	***COM1***	***COM2***	***COM3***
Significant	*alteration in lab values*	*Elevation in liver function*	*test*	
Patient non-compliance		*Patient relocated to other*	*country. Unable to*	*continue study*

By using macro variables to represent the number of variables, code can be written which is flexible enough to handle any number of variables. The code presented below uses %LET statements to assign values to the macro variables so that a programmer need only make changes in 1 place when the number of variables changes. It is possible to write code which will assign these values automatically, but that code is beyond the scope of this paper. The variables can be combined into a series of variables (COMT1-COMT5) using this code:

```
%LET REASN=2;
%LET COMN=3;
%LET COMTN=%EVAL(&REASN + &COMN);

DATA COMT; SET COMT;
      ARRAY COMT {*} $22    COMT1-COMT&COMTN;
      ARRAY REAS {*}        REAS1-REAS&REASN;
      ARRAY COM  {*}        COM1-COM&COMN;
      CNT=0;
      DO I=1 TO &REASN UNTIL(REAS(I)='');
            J=I+CNT;
            IF I=1 & REAS(I)^='' THEN COMT(J)='R:'||REAS(I);
                ELSE COMT(J)='  '||REAS(I);
      END;
      CNT=J-1;
      DO I=1 TO &COMN UNTIL(COM(I)='');
            J=I+CNT;
            IF I=1 AND COM(I)^='' THEN COMT(J)='C:'||COM(I);
                ELSE COMT(J)='  '||COM(I);
      END;
      DROP I J CNT REAS1-REAS&REASN COM1-COM&COMN;
RUN;
```

In this code we are using 3 arrays, COMMT, REAS AND COM. The first DO loop places all of the text from the REAS array variables into the first COMMT variables. We check for REAS(I) not equal to missing because we don't want to create a missing variable in the middle of the array. We want all the text to print on consecutive lines, and creating a missing variable will cause a blank line to print within the block of text. The second DO loop does the same thing for all the text from the COM array. Notice also that we append a single letter to the first line of each to indicate if it is a reason (R) or comment (C). We then indent all remaining lines to make the output easier to read. Finally, we drop all the extra variables from the output data set. The resulting SAS data set will look like:

COMT1	COMT2	COMT3	COMT4	COMT5
R:Significant	alteration in lab values	C:Elevation in liver function	test	
R:Patient non-compliance	C:Patient relocated to other	country. Unable to	continue study	

We can then use arrays in the DATA _NULL_ to make writing the PUT statements easier:

```
DATA _NULL_; SET COMT END=EOF;
      FILE PRINT .....
      •
      •
      PUT ..... @80 COMT1;
      ARRAY COMT {&COMTN};
      DO C=2 TO &COMTN WHILE(COMT(C)^='');
            PUT @80 COMT(C) $CHAR22.;
      END;
```

The DO WHILE loop here will execute the PUT statement only if the COMT array element is not missing. This example also shows a different format for the ARRAY statement. If the elements of an array are a numbered list of variables (i.e. COMMT1, COMMT2, COMMT3, etc.) the do not need to be listed if the array name is the base name of the elements (COMMT in this example). By specifying the array name and the subscript, SAS will automatically create the elements as a numbered list with array_name as the based. The output report will look like:

```
                Reason for Termination(R)
                Comment(C)
                R:Significant
                   alteration in lab values
                C:Elevation in liver function
                   test

                R:Patient non-compliance
                C:Patient relocated to other
                   country. Unable to
                   continue study
```

Example 4: Using temporary variables in arrays

Many times we have a need to create variables for processing in data step, but don't want to include those variables in an output data set. Use of a DROP or KEEP statement will eliminate unwanted variables, but use of these statements can be avoided in some instances. If you are processing a group of these variables in the same manner, you can use temporary variables in an array.

Suppose that we have created a data set from a PROC CONTENTS which contains the names of all members of a SAS library and want to string those names together in a series of macro variables for use in another part of the program. The code below shows how this can be done using an array with temporary variables.

```
DATA ALLDAT;
      SET ALLDAT END=EOF;
      ARRAY NAMES {2} $200 _TEMPORARY_;
      RETAIN X 1;
      NAMES(X)=TRIM(NAMES(X))||''||TRIM(MEMNAME);
      IF LENGTH(NAMES(X))>190 THEN DO;
            X+1;
            IF X>2 THEN DO;
               PUT "*** YOU HAVE MORE DATA SET NAMES ***" /
                     "*** THAN WILL FIT INTHE ARRAY USED. ***" /
                     "*** INCREASE ARRAY DIMENSION AND  ***" /
                     "*** MODIFY CALL SYMPUT             ***" /
                     "*** STATEMENTS BELOW THEN TRY      ***" /
                     "*** AGAIN.                         ***";
               ABORT RETURN;
            END;
      END;
      IF EOF THEN DO;
            CALL SYMPUT('DSNAMES1',NAMES(1));
            CALL SYMPUT('DSNAMES2',NAMES(2));
      END;
RUN;
```

We set up an array of temporary variables by using the _TEMPORARY_ keyword in the ARRAY statement. Since we don't know how many members we will be using and the length of their names, we set these variables to the maximum length allowed, 200 characters. We retain the value of X between passes through the data set so that we are working with the same array element until it is full. Next, we add the member name from the current observation by concatenating it to the current array element. We then check to see if the current element is full. Since we know that the maximum length for a SAS data set is 8 characters and that we need 1 space between the member names, we will need at most 9 characters in the array element to add the next member name. We have rounded this off to 10 (200-190) in the IF statement where we do the checking. If the length of the current element is greater than 190, we execute a DO loop where we increase X by 1 so that we will process the next array element on the next pass through the data set. We then check to see if X is greater than the number of elements in our array. If it is, we print a message to the log and abort the program. This is done because if X becomes greater than the array the data step will stop executing, an error message will be written to the log and the SAS program will stop executing, but continue with syntax checking. Our reason for aborting the program is to save processing time when an error occurs. The last DO loop executes only when the data step reaches the end of the input data set and creates the macro variables we want.

Conclusion

We hope that we have taken some of the mystery out of SAS arrays and their use. Arrays are a powerful tool that can make a programmers life much easier. They can do this by reducing the lines of code need to perform a task; allowing you to use code and procedures which are easy to understand; and making the modification of programs easier by reducing the number of statements you need to modify. We hope that you will consider using arrays in the future to make your life easier.

Reference

SAS Institute, Inc., (1990), *SAS Language Reference Version 6 First Edition*, Cary, NC, SAS Institute, Inc.

Author Contact

Greg M. Woolridge
Searle
4901 Searle Parkway
Skokie, IL 60077
greg.m.woolridge@monsanato.com

Winnie Lau
ASG, Inc.
175 E. Hawthorn Parkway, Suite 155
Vernon Hills, IL 60061
wll@asg-inc.com