Paper 216-2007

Everyone Needs a Raise (Arrays)

Marge Scerbo, National Study Center, UMB

Abstract

Beginning SAS programmers often write much longer code just to avoid using arrays. Arrays have an aura of difficulty, but they are simply misunderstood and underutilized.

This tutorial will introduce you to the basics of SAS Arrays and provide examples of different ways and times to use these data step statements. By utilizing arrays, you can process multiple variables in an efficient manner.

Hopefully, you will feel confident enough when the talk is over to return to your office and put your new knowledge to use.

Introduction

Some programmers actually like writing lines and lines of simple code. As long as the project is completed and the results are delivered within the expected time frame, this is usually acceptable. Other programmers hate to type and find any method possible to write as little code as possible, but similarly want to get the job done. In either case, code may be easy or hard to review. A very long monotonous program may be difficult to debug simply because of volume; fix one mistake and another one is discovered. A very terse program with complicated algorithms may also be difficult to interpret by either the original or subsequent programmer.

SAS code allows programmers to write either way, long or short, easy or complicated. It is possible to get to the same result from different directions. **Data** step programming is the core to SAS code. Within a **data** step, it is possible to accomplish many different tasks. This paper will add one technique, the **array** statement, to a programmer's toolbox. In addition to an overview of the **array** statement, several basic and intermediate examples of array usage will be presented. Although SAS arrays may be multi-dimensional, this tutorial will discuss only one-dimensional arrays.

Long Way Around -- Example #1

The best way to show how arrays are useful is by demonstrating the two different methodologies. The first example uses **data** step code without arrays and the second method includes **array** processing.

This company, Nice Guy Inc., bases annual raises on a series of rating scores. The evaluations are completed by the managers, and then results are entered into the system. The scores from the previous year must be reset to zero before this process can begin. There are 14 scores in this schema, and a simple **data** step can be written to handle this problem:

```
data nonarray1;
       set employees;
       score1  = 0 ;
       score2  = 0 ;
       score3  = 0 ;
       score4  = 0 ;
       score5  = 0 ;
       score6  = 0 ;
       score7  = 0 ;
       score8  = 0 ;
       score9  = 0 ;
       score10 = 0 ;
       score11 = 0 ;
       score12 = 0 ;
       score13 = 0 ;
       score14 = 0 ;
run;
```

This code is correct and meets the requirements, but can you imagine maintaining the same code 400 different scores!

Basic Array Syntax

An **array** statement must 'exist' in a SAS data step. It does not function within a procedure. The basic format of an **array** is:

***Array* array-name(number-of-elements) array-elements;**

The pieces of the ***array*** puzzle include:

- An **array-name** identifies the group of variables in the array.
- The **number-of-elements** identifies the number of elements (variables) in a one-dimensional array.
- The **array-elements** are the elements or fields that make up the array.

The **array-name** cannot be the name of a SAS variable used in that data step. In other words, it cannot be the name of any variable in the data set(s) read in or output. Although there are no errors encountered if the array-name is the name of a SAS function, it is dangerous to use a function-name as the array-name. The results may not be valid.

The **number-of-elements** can be either a number or numbers, a calculation, a numeric variable, or an asterisk '*'. Again, the **number-of-elements** designates how many elements exist in an array. If the number of elements is unknown, using the asterisk, '*', allows SAS to count the number of elements. This subscript is always enclosed in parentheses ( ), brackets [ ], or curly braces { }. SAS functions use parentheses, so to avoid using the parentheses incorrectly, you could choose to use brackets or braces.

Arrays can be either numeric or character but not a combination of types. If the array is character, the subscript is followed by a dollar sign '$'. The subscript and, if needed, the dollar sign designating a character array can be followed by a number that assigns the length of each element.

Array-elements are a list of the variable names. Again, these must be all character or all numeric fields. These are variables that are referred to as **array-name**(subscript number) during processing. If no **array-elements** are named, SAS creates new variables that are named array-name with the subscript number concatenated to the end. For example, if you name a new array *newvar* with 50 elements and allow SAS to create the array elements, they will be called *newvar1* to *newvar50*.

The **array-name** and all the **array-elements** must be valid SAS names. SAS Version 8 and higher allows names to be between 1 and 32 characters long, beginning with a letter (A-Z) or an underscore (_). These names cannot contain a blank or any special character except the underscore. Finally, names cannot be SAS reserved words.

Shorter way around - Example #1

The first example provides a perfect use for arrays. Remember, this process is resetting the values of variables *score1* through *score14* to a value of 0:

```
data array1;
       set employees;
       array scores(14) score1-score14;

       do i = 1 to 14;
              scores(i) = 0 ;
       end;
       drop i;
run;
```

To further identify the array parts:

- the **array-name** is *scores*
- the **number-of-elements** is 14
- the **array-elements** are *score1* through *score14*.

Note that the number of variables (*score1* through *score14*) equals the subscript (14). The **do loop** is executed 14 times and the **index-variable** *i* is incremented from 1 to 14. At each increment, the **array-element** value is set to 0. Before exiting the data step, the **index-variable**, *i*, is dropped, as it does not need to be stored in the data set.

To step through the first increment of the **do loop**:

- the **index-variable** *i* is set to 1
- *scores(1)* is set to 0 therefore score1 = 0
- the **end** of the **do loop** is encountered and the process begins again with the **index-variable** incremented to 2

If the situation were such that there were many elements involved, creation of an array statement that used SAS code to count and reset the values would be no more difficult:

```
data arraybig1;
       set bigbasefile;
       array score{*} score1-score400;

       do i = 1 to dim(score);
              score(i) = 0 ;
       end;
       drop i;
run;
```

This data step defines an **array** called *score*. By using the asterisk '*', SAS will count the number of array variables *score1* to *score400*. The **do loop** is set to execute from 1 to the extent of the array *score*. The **dim()** function counts the array elements. So for example, if Nice Guy Inc. changed the number of scores annually, the only change to this program would be the maximum number of scores, in the above example *score400*.

Long way around -- Example #2

The last example involved variables with the same base name, *score*. It is very probable that some projects require manipulation of a list of variables with a variety of names. In this next case, there is a need to recalculate a list of payments to include a cost-of-living increment if the amount is greater than 0. There are 7 fields, and the non-array code would be:

```
data nonarray2
       set basefile2;

       if basepay gt 0    then basepay    = basepay * 1.0345;
       if copay gt 0      then copay      = copay * 1.0345;
       if fedpay gt 0     then fedpay     = fedpay * 1.0345;
       if insurepay gt 0  then insurepay  = insurepay * 1.0345;
       if deductible gt 0 then deductible = deductible * 1.0345;
       if savepay gt 0    then savepay    = savepay * 1.0345;
       if pretaxpay gt 0  then pretaxpay  = pretaxpay * 1.0345;
run;
```

Of course, this does work. If new fields are to be included, then more lines need to be added, and so on.

Shorter way around - Example #2

The last example can be written in **array** code. In the example below, *pay(1)* equals *basepay, pay(2)* equals *copay*, and so on. All variables are numeric; you cannot combine character and numeric fields in an array. An **array** must contain all numeric or all character fields!

```
data array2;
      set basefile2;
      array pay(7) basepay copay fedpay insurepay deductible savepay
                   pretaxpay;
      do i = 1 to 7;
           if pay(i) gt 0 then pay(i) =  pay(i) * 1.0345;
      end;
      drop i;
run;
```

If new variables were added to this process, they would be added to the list of **array-elements** and the **subscript** and **do loop** counter would be changed.

A Short Character Example

In this example, the managers need to count the number of times within years from 2001-2007 that each employee received a raise. This will allow the managers to review the work habits of those employees that have not received any raise during this time, assuming they have been employed for at least 5 years.

The database contains many years worth of data. There are annual fields called raise followed by the year, for example *raise2004*. These character fields are set to Y or N. The study reports those employees that during the last 7 years have not received a raise for at least 5 of those years.

```
data nonarray3;
       set employees;
       noraise = 0;

       if raise2001 = 'N' then noraise + 1;
       if raise2002 = 'N' then noraise + 1;
       if raise2003 = 'N' then noraise + 1;
       if raise2004 = 'N' then noraise + 1;
       if raise2005 = 'N' then noraise + 1;
       if raise2006 = 'N' then noraise + 1;
       if raise2007 = 'N' then noraise + 1;
run;
proc print data = nonarray3;
       var employee noraise;
       where noraise ge 5;
       title 'At least 5 of 7 years with no raises';
run;
```

The same results can be obtained by using an **array** statement. In this case, we are creating a character **array** called *raises* and so the **array** statement includes a dollar sign '$'.

```
data array3;
       set employees;
       noraise = 0;
       array raises(7) $ raise2001-raise2007;
       do i = 1 to 7;
              if raises(i) = 'N' then noraise + 1;
       end;
       drop i;
run;
```

```
proc print data = array3;
       var employee noraise;
       where noraise ge 5;
       title 'At least 5 of 7 years with no raises';
run;
```

Again, the code is similar to the numeric array example. Since there are only 7 years involved, there are no real differences between the non-array and array examples. If more years were added, the non-array code would get longer and longer, and errors could easily occur. It is quite easy to copy the line of code over and over, but remember that the number attached to the variable name must be changed, and it is simple to miss one number or repeat a number.

An Multi-Array Example

This example is based on some of the work we have already discussed in the paper. Assume that each employee record contains a variable called annual*nnnn* that is the annual raise per year (*nnnn*). The values are stored as percentages of a variable called *basepay*. As shown in example 3, there are character fields (*raise2001-raise2007*) that show whether or not the employee received a raise.

In this example, we need to calculate the total dollar amount that each person received in raises across these 7 years. First we will check to see if the employee received a raise and if so, we will calculate their total raise by summing up the dollar amounts across years.

In this example of non-array versus array code, the specifications for this process include:

- This employee file includes several repeating fields. Two groups of these fields are raise codes (*raise2001-raise2007*) and annual percentages (*annual2001-annual2007*).
- The variables *raise2001-raise2007* are character, and *annual2001-annual2007* are numeric.
- This total dollar amount is to be calculated for each year the employee received a raise, based on a variable called *basepay*.
- The result of this process is the total dollar amount (*totraise*) of raises across the 7 years.

The non-array code looks like this:

```
data nonarray5;
   set employees;
   totraise = 0;

   if raise2001 = 'Y' then totraise = totraise + (annual2001 * basepay);
   if raise2002 = 'Y' then totraise = totraise + (annual2002 * basepay);
   if raise2003 = 'Y' then totraise = totraise + (annual2003 * basepay);
   if raise2004 = 'Y' then totraise = totraise + (annual2004 * basepay);
   if raise2005 = 'Y' then totraise = totraise + (annual2005 * basepay);
   if raise2006 = 'Y' then totraise = totraise + (annual2006 * basepay);
   if raise2007 = 'Y' then totraise = totraise + (annual2007 * basepay);
run;
```

We can create the same results using two arrays, *raises* that is character and *annuals* that is numeric.

```
data array5;
       set employees;
       totraise = 0;

       array raises(7) $ raise2001-raise2007;
       array annuals(7) annual2001-annual2007;
```

```
        do i = 1 to 7;
              if raises(i) = 'Y' then
                    totraise = totraise + (annuals(i) * basepay);
        end;
        drop i;
run;
```

As these examples get more complicated, the efficiency of array programming becomes more evident!

Simple Input Example

Arrays can be useful during the creation of a data set. There may be instances when the file layout contains fields that would be read in the same order and with the same specifications, except further along the line of data.

These examples use input pointer control. A **pointer-control** input statement includes an at sign '@' followed by the column specification, the variable name, and the informat of the field. This column specification can be a number, a calculation, or a numeric variable.

Each year Nice Guy Inc. surveys their employees. This survey includes five ratings of their supervisor. This data is entered on a survey sheet by employees, fed through a scanner, and stored initially as a text file. Each rating code is one or two characters long. The non-array code to read this data could be:

```
data annualreviews;
      infile 'annualreview.dat' lrecl = 112 missover;

      input  @1   employeeid     $11.
             @12  reviewdate     mmddyy10.
             @40  supervisorid   $11.
➔            @100 rating1        $2.
             @102 rating2        $2.
             @104 rating3        $2.
             @106 rating4        $2.
             @108 rating5        $2.  ;
run;
```

Note that this group of 5 ratings begins in ➔column 100, and each rating is 2-characters long. The new code using an array to complete the task above follows:

```
data annualreviews;
      infile 'annualreview.dat' lrecl = 112 missover;

      array rating(5) $2;

      input  @1    employeeid           $11.
             @12   reviewdate           mmddyy10.
             @40   supervisorid         $11.
             @100  (rating1-rating5)    ($2.)  ;
run;
```

So, to parse the various pieces of code in this example:

- A **character** ($) **array** *rating* is built with 5 elements, corresponding to the 5 survey ratings.
- The **array-elements** are automatically defined as variables *rating1* through *rating5*. As shown, since no specific array elements are listed, the elements are named *arrayname1* through *arrayname5*.
- The fields, *employeeid*, *reviewdate* and *supervisorid* are read.

- The pointer is moved to column 100 (**@100**), and each of the rating codes is read in sequence (**rating1-rating5**) and each as a character field two characters long (**$2.**).
- If additional fields are to be read, the variable definitions can be added to the **input** statement.

This example clearly demonstrates how very little code can be used to perform this operation.

Simple Output Example

In preparing data for certain types of analysis, it is sometimes best to reconfigure the fields to be searched, assuming there are a group of variables in one record that contain similar data. In other words, when searching for a set of values, it may be easier to search down the data set rather than across the records. This process converts a short fat data set into a long thin one.

Take the example of the data set just created. In order to search for managers with high ratings, it is possible to recreate the data set as one long and narrow data set, containing one rating per record and whatever other fields may be needed. In following example, there are only four values that constitute good ratings (EX, SP, GD, SA).

In this example, a program is written which reviews each of the 5 rating fields. If a 'good' value is encountered, a record containing that rating, the supervisor id, and the date is output to a new data set. If a missing value or bad rating is encountered, the process continues on to the next rating. The report will not include any bad ratings for a supervisor.

Without using an array statement, the code might be written as:

```
data bestreview06 (keep = supervisorid monthreview rate);
       set annualreviews;
       length rate $2;

       if rating1 in('EX','SP','GD','SA') then do;
          rate = rating1;
          output;
       end;
       If rating2 in('EX','SP','GD','SA') then do;
          rate = rating2;
          output;
       end;
       if rating3 in('EX','SP','GD','SA') then do;
          rate = rating3;
          output;
       end;
       if rating4 in('EX','SP','GD','SA') then do;
          rate = rating4;
          output;
       end;
       if rating5 in('EX','SP','GD','SA') then do;
          rate = rating5;
          output;
       end;
  run;
```

Using an array, the data step could be:

```
data bestreview06 (keep = supervisorid monthreview rate);
       set annualreviews;
       length rate $2;
```

```
        array rating(5) $;
        do i = 1 to 5;
              if rating(i) in('EX','SP','GD','SA') then do;
                    rate = rating(i);
                    output;
              end;
        end;
run;
```

The **in** operator provides a list of values to be tested. In the above case, the value of the rating should be **in** that list. If a bad rating is encountered, the record is not included in the report.

Reporting with Arrays

In almost any industry, the data are only as valuable as the reports that can be generated from it. Creating complicated reports is a time-consuming job. Arrays can be used in a variety of ways to help create both the summarized and/or subset data sets and the final reports.

In the example below, an array is used to create a string of rating codes. Note that the data set accessed (*annualreviews*) is the same data set manipulated in the last example. This large file contains information on manager ratings, including up to five employee-surveyed ratings in one record. The report to be created is to contain the supervisor id number (*supervisorid*), the date of review (*monthreview*), the number of negative rating codes identified (*ratecnt*), and a list of codes (*codelist*) separated by commas in the format:

rating code 1,rating code 2, …, rating code 5

The report will not contain 'good' or missing ratings, only those codes that do not meet the stated standards. The following code creates an output data set. In production mode, this step could be combined with the actual report creation.

```
data badreport (keep = supervisorid monthreview ratecnt codelist);
      set annualreviews;
      array ratings(5) $2 rating1-rating5;

      *set the length of the final ratings list;
      length codelist $20;

      ratecnt = 0;
      do i = 1 to 5;
         if ratings(i) not in('EX','SP','GD','SA',' ') then do;
             ratecnt = ratecnt + 1;
             if ratecnt = 1 then codelist = ratings(i);
             else codelist =  left(trim(codelist)) || ', ' || ratings(i);
         end;
      end;
run;
```

This code uses the character functions **left** and **trim** and the concatenation operator **||** to create the list in the correct format. Within the **do loop**, these operations occur:

- The code counter (*ratecnt*) is incremented each time a bad rating code is encountered.
- When the first accepted code is identified, the list of codes (*codelist*) is set to this code.
- Any additional valid codes are added to the list string, which is left justified (**left**), trimmed of trailing blanks (**trim**), and concatenated (**||**) with a comma (**,**) and a blank space.

- In version9, the CATX function will perform this same process within one function. See Ron Cody's Character Functions paper and his 'SAS Functions by Example' SAS Press book for more information.

The output data set (*badreport*) can now be used to create a complete report of bad ratings for each supervisor.

Calculated Subscripts

Data are often stored in 'annual' files containing all the data for a specific period. In this example, the data files are arranged according to fiscal year July 1 through June 30. In order to do calendar year calculations, creation of a new data set is needed.

Each fiscal year file is sorted by an employee identifier (*employeeid*) and contains 12 numeric variables, *comm1* to *comm12*, where *comm1* equals the commission for July and *comm12* equals the commission for June. To create the 2006 calendar year file, commissions from January through June (*comm7-comm12*) from fiscal year 2005 and July through December (*comm1-comm6*) from fiscal year 2006 are needed. This process will calculate the annual commissions (*totcommission*) for each employee.

See the tables below. They show how the variables exist in the fiscal year (FY) files and how they will be redefined in the calendar year (CY) file.

Beginning Datasets – FY05 and FY06

FY05	*comm7*	*comm8*	*comm9*	*comm10*	*comm11*	*comm12*
	Jan	Feb	Mar	Apr	May	Jun
FY06	*comm1*	*comm2*	*comm3*	*comm4*	*comm5*	*comm6*
	Jul	Aug	Sep	Oct	Nov	Dec

Ending Dataset – CY06

CY06	*newcomm1*	*newcomm2*	*newcomm3*	*newcomm4*	*newcomm5*	*newcomm6*
	Jan	Feb	Mar	Apr	May	Jun
	newcomm7	*newcomm8*	*newcomm9*	*newcomm10*	*newcomm11*	*newcomm12*
	Jul	Aug	Sep	Oct	Nov	Dec

The resulting data set should contain new variables *newcomm1* to *newcomm12* in the correct sequence, where the January commission is *newcomm1*, etc.

```
data cy06;
        merge fy05 (keep = idnum comm7 - comm12)
              fy06 (keep = idnum comm1 - comm6);
        by idnum;

        totcommission = 0;
        *array to create new variables;
        array newcomm(12) newcomm1 - newcomm12;

        *arrays contain fields from fiscal year files;
        array fy05(6) comm7 - comm12;
        array fy06(6) comm1 - comm6;

        do j = 1 to 12;
              if j le 6 then newcomm(j) = fy05(j);
              else newcomm(j) = fy06(j - 6);
              totcommission + newcomm(j);
        end;
        drop j comm1-comm12 ;
run;
```

Note that the subscript (*j*) is a calculated field for the array *fy06*. While the subscript for the array *newcomm* ranges from 1 to 12, the subscripts for *fy06* range from 1 to 6. Therefore, the subscript is equal to the counter minus 6 (*j - 6*).

Dates and Arrays

The following example uses a SAS date function and an array to satisfy a request. This company, Nice Guys Inc. gives regular bonuses to its employees. For new employees, the bonus is normally given one year from the start month. This next program was written to check that the new employees did indeed receive a bonus.

Assume a data set contains 12 flags (*janbonus06-decbonus06*) indicating the bonus amount an employee may have received during that month. If the month field contains a 0, the employee did not receive any extra funds. The management needs a list of new employees who did not receive their expected bonus.

```
data newemployees (keep = employeeid startdate);
       set employees;
       where year(startdate) = 2006;

       array bonus(12) janbonus06--decbonus06;

       *the month function identifies the starting month;
       mon = month(startdate);

       if bonus(mon) = 0 then output;
run;
```

What could have been a very complicated task was made easy through the use of an array! Note that in this example there is no **do loop** to review all the elements in the array. The code merely queries the month in question.

Arrays to Eliminate Unwanted Observations

There are times when it may be necessary to eliminate observations from a specific study. These deletions may be due to very specific requirements of the analysis. One reason might be that there are too many missing values in the observation. Of course, code can be written to test each variable to determine if the value is missing or not and ascertain whether or not to include the observation. The example below contains a generic approach to this situation.

```
data complete;
       set maybeokay;
       misscount = 0;

     array  ch(*) $  _character_ ;
     array  nm(*)  _numeric_ ;

     do i = 1 to dim(ch);
       if ch(i) = ' '  then misscount = misscount + 1;
     end;
     do j = 1 to dim(nm);
       if nm(j) = .   then misscount = misscount + 1;
     end;
     if misscount ge 10 then delete;
     drop i j misscount;
run;
```

Note that this SAS code contains two special SAS variables that can be used in the program. These variables do not exist in the dataset:

- **_character_** is a special SAS variable that identifies and includes only character variables in the process described
- **_numeric_** similarly identifies only numeric fields

In using this code, the program can process across almost any data set without requiring the program to include an exact count of the array elements, be they character or numeric.

One other SAS special variable you may find useful is **_temporary_**. This variable will allow you to create an array that is temporary and therefore only exists during that particular data step. This can be useful if you are doing calculations based on certain array elements and the final outcome is needed, not the individual elements

Conclusion

With a little practice and common sense, arrays can become a standard tool in a programmer's tool belt. Follow these tips:

- First, always have a SAS Language Guide, either paper or online, available!
- In the process of learning how to use arrays, make sure to test the program with non-array code. Print out the Log and Output.
- Then rework the program to include array code and compare these results with the non-array code.

After a while, it will become second nature to use arrays. Once the learning curve is over, the usefulness will increase and soon there will be multiple arrays and do loops within do loops!

Contact Information

For more information contact:
Marge Scerbo
National Study Center
701 W. Pratt, Room 554
Baltimore, MD 21201
Email: mscerbo@som.umaryland.edu