A05 - 2008

Loop-Do-Loop Around Arrays

Wendi L. Wright, CTB McGraw Hill, Harrisburg, PA

ABSTRACT

Have you ever noticed your data step repeats the same code over and over? And then thought … there must be a better way. Sure, you could use a macro, but macros can generate many lines of code. Arrays, on the other hand, can do the same job in only a few lines. Many SAS® programmers avoid arrays thinking that they are difficult, but the truth is they are not only easy to use, but make your work easier.

Arrays are SAS data step statements that allow iterative processing of variables and text. We will look at many examples, including 1) input and output of files using arrays, 2) doing the same calculation on multiple variables, and 3) creating multiple records with one observation. This tutorial will present the basics of using array statements and demonstrate several examples of usage.

INTRODUCTION

There are many ways that arrays can be used. They can be used to:

- Run the same code on multiple variables (saves typing)
- Read variable length input files
- Create multiple records from one observation
- Create one observation from multiple observations

Arrays in SAS are very different from arrays in other programming languages. In other languages, arrays are used to reference multiple locations in memory where values are stored. In SAS, arrays may be used for this purpose (temporary arrays), but they also may be used to refer to lists of variables (the most common use of arrays in SAS). This allows the programmer to assign a value to a variable without knowing what the variable name is, an extremely useful tool.

RULES OF USING ARRAYS

In order to use arrays correctly, there are several things you need to keep in mind:

- All variables assigned to an array must be the same type – either character or numeric.
- The array name itself is temporary and so is not available outside the data step. However, the variables the array represents are not temporary and so can be used in procedures and other data steps.
- If you reference an array with a non-integer index, SAS will truncate the index to an integer before doing the array lookup.
- If the array is initialized, all the variables are retained (even if only some are initialized).

Arrays are very flexible:

- Variables in an array may or may not exist.
- Array names assigned to an array follow regular variable name restrictions.
- Variables can be blank or not (or can be initialized or not).
- Variables can be different lengths (particularly useful when using character variables).
- Any variable may be in more than one array.
- Arrays can be extremely large. Note: One of the significant points about using _TEMPORARY_ arrays in version 6 was that you could have arrays with more elements than the maximum number of allowed variables in a data step. That has changed now with version 8 and 9.
- Arrays can be multidimensional. I successfully tested an array with 10 dimensions before running out of memory on my PC. The limit is most likely based on how much memory is available on

your platform.

SYNTAX OF ARRAYS

ARRAY *arrayname* [3] $ 2 var1 – var3 ('H4' 'J6' 'K3');

Here is an example of an explicitly defined array statement. Let's break down this statement into its parts.

The *arrayname* can be anything you want up to 32 characters. Array names **cannot** be the same as any of your variable names. They **can** be the same name as a SAS function, and they will override the function when used in code.

The [3] in brackets tell how many variables you want this array to hold. The brackets can be parentheses ( ) or squiggly brackets { } as well.

The history of this is interesting to note. Parentheses ( ) were used on the IBM mainframe and later, when SAS ported to VAX, there was a problem with parentheses, so [ ] were used instead. Then to satisfy user complaints about portability, { } were added. Today, all platforms accept all three versions in the array statement, so use your preference.

The $ 2 says these elements are character variables with a length of 2. The '$' is necessary if these variables have not previously been created. If you are loading previously defined character variables, then you do not need to specify the variable type. If you specify a different length for variables than already exist, SAS ignores the length specified on the array statement. For new variables, if you don't specify a length, the default is 8.

Var1-var3 are the variable names to be included in this array. You can specify the list with or without the dash(es). The double dash can be used in the array statement for those of you familiar with its use.

('H4', 'J6' and 'K3') are the initial values that will be placed in these variables for EVERY observation. Note that in an array, if the variables are initialized, they are retained. These values are what is written to the output dataset unless you specifically change them during processing.

Here are a few examples of valid array statements:

Array Quarter {4} Mar Jun Sept Dec (0 0 0 0);	Numeric array with initial values. Variable names are Mar, Jun, Sept and Dec.
Array Days {7} $20 d1 – d7;	Character array no initial values. Variable names are d1, d2, etc to d7. Each variable has a length of 20.
Array Month {6} jan -- jun;	Array with six members assigned the variables from jan to jun.

NEAT FEATURES AND TRICKS TO DEFINING ARRAYS

A neat feature of arrays is that SAS can count the number of variables. To have SAS do the counting, put a * in the brackets instead of the number of elements. SAS will count the number of variables and then define the array with that number. We will look at how useful this can be in some of the examples later in this paper.

SAS can count the number of array members	Same as
Array Quarter {*} Jan Feb Mar;	Array Quarter {3} Jan Feb Mar;

SAS can create the variable names for you. If the variable names are to be consecutively named, like month1 through month12, then you can define the array with just the character portion of the name and the number of members.

SAS can assign the variable names.	Same as
Array Month {12};	Array Month {12} Month1-Month12;

Note you cannot tell SAS to count the number of members and also create the member names.

CANNOT USE: Array Item {*};

SAS also has a few code words to put all the existing character or numeric variables into an array for you. These can save a lot of typing. This is useful for cleaning up missing values in your variables. Note: If you use the _all_ code word, you need to be sure that all the variables in your data step must be either numeric or character. This is necessary because all the variables assigned to an array must be the same type – either character or numeric. If not, SAS will return an error.

Array char {*} _character_;
Array Days {*} _numeric_;
Array Month {*} _all_ ;

ASSIGNING INITIAL VALUES

Let's look more closely at assigning values to the array members. The rules for creating array values are:

- Values must be specified within parentheses.
- Values must always be specified LAST in the array statement.
- Character values need to be in quotes.
- Values may be given to some or all of the members of your array.
- Iteration factors can be used to repeat all or some of the initial values.
- All variables that have been initialized will be retained.

The following example creates six test name variables and initializes the first four. The other two variables are not initialized. All will be retained across all observations. Note the length was not specified (only the $ sign used), so the default length of eight is used in the creation of the variables.

Array newvars {6} $ test1 – test6 ('English' 'Math' 'Sci' 'Hist');

The next set of examples are all equivalent and show the use of iteration factors and nested sub lists. Note for the second and fourth examples we are asking SAS to create the member names for us and for the third example we are asking SAS to count the number of members.

Array a {10} a1-a10 (3 3 3 3 3 3 3 3 3 3);
Array a {10} (10 * 3);
Array a {*} a1-a10 (5 * (3 3));
Array a {10} (2*3 2*(3 3 3) 3 3);

REFERENCING ARRAYS

We've looked at how to assign arrays, but how do we tell SAS to look up the array values?

To reference an array member, use the name of the array and then in either brackets or parentheses, place the number of the member you want to reference. By default, SAS starts the numbering system at 1 and moves up by one for each member. This is different than other languages, such as Java and VB, which start their array numbering system at zero.

In the following example, an array newvars has three elements, called test1, test2, and test3 with the values of 45, 23, and 21, respectively. If you provide a non-integer value to reference an array element, SAS will truncate the number to an integer. Now that you know how to reference an array element, it is very easy to take it one step further and change the array element. Just use the array reference on the left hand side of an assignment statement.

Array newvars [3] test1-test3 (45 23 21);
Newvars[3] will return 21
Newvars[2.22] will return 23

Taking it even one step further, you can embed the assignment statement within an IF statement or a DO loop. In the example below, we are changing the first array member or variable 'key1' to a 'B' if the second element in the newvars array matches a specific value. This example demonstrates that an array reference can be used in both the comparison and the action side of an IF statement.

Array key [5] $ key1-key5;
If newvars(2) eq 23 then key[1] = 'B';

ASSIGNING YOUR OWN SUBSCRIPTS

By default, SAS assigns the subscripts for an array starting with 1 and adding one incrementally up to the number of members in the array. It is possible to override this. To specify the starting and ending reference values in an array, specify the range numbers you want to use separated by a colon inside the brackets in the array definition statement.

Note: These are NOT two-dimensional arrays. Two-dimensional arrays separate the number of columns and rows with a comma, not a colon. Two dimensional arrays are discussed later in this paper.

To reference these arrays, you do NOT specify years(1), you specify years(2000), or years(2001). Here are two examples.

Example 1	Example 2
Array years [2000:2004] $ revenue1 – revenue5;	Array items [23:25] item23 – item25 (‘B’ ‘A’ ‘C’);
Years(2003) would return value from revenue4.	Items[25] would return ‘C’.
Years(2) would return an error.	Items[1] would return an error.

THE DO LOOP

Because arrays are easily referenced with an index value, they are very often used with a do loop. There are three types of do loop.

The first type of DO loop uses a list (either numeric or character) and the loop is executed once for each value in the list specified in the do loop. One common use is to provide a start value, end value, and incrementation factor (if the incrementation factor is not provided, then SAS assumes the value is one). Here are a couple of examples:

Do i = 1 to 25 by 1;
Do i = ‘A’, ‘B’, ‘C’;

The two other types of do loops available are the Do Until and the Do While. The differences between the two loops are 1) when the condition is tested and 2) whether the condition needs to be true or false for the loop to end. Do Until is tested at the bottom of the loop and the condition needs to be true to stop the loop. This means that the code inside is executed at least once, even if the condition is true when the loop starts. The Do While loop is tested at the beginning of the loop and the condition needs to be false to stop the loop.

Loop Type	When condition is tested	What condition stops loop
Do Until (condition);	at bottom	A true condition
Do While (condition);	at top	A false condition

All three types of loops need an ‘END;’ statement at the end of the block of code.

The different types of do loops can be combined into one. In the examples below, SAS will cycle through the iteration factor until the condition is met and then it will end the loop.

Do i = 1 to 25 Until (condition);
Do i = ‘A’, ‘B’, ‘C’ While (condition);

EXAMPLE 1 – INITIALIZING VARIABLES

We have 62 variables (item1-item60, tax, and total) that we want to check for blanks and if we find them, change them to zeros.

Before the use of arrays, we would need to do:

```
If item1 eq . then item1 = 0 ;
If item2 eq . then item2 = 0 ;
If item3 eq . then item3 = 0 ;
….
If tax eq . then tax = 0 ;
If total eq . then total = 0 ;
```

This would be 62 lines of code!! So let's try to shorten it up a little. Here's how to do it with a macro:

```
%macro check;
        %do i = 1 to 60;
                if item&i eq . then item&i = 0;
        %end;
%mend;
%check
if tax eq . then tax=0;
if total eq . then total=0;
```

However, the macro *still* feeds 60 lines of code to the data stack and we *still* had to process two of the variables (tax and total) outside of the macro. If we had all different named variables, we could not have used macros at all!

Using arrays is the answer. The same thing can be done with only 4 lines of code. And we don't have to do the check for tax and total separately.

```
Array  var  (62)  item1-item60  tax total;
Do  i  =  1 to 62;
        if  var{i} eq . then var{i}  = 0 ;
End;
```

EXAMPLE 2 – PERFORMING CALCULATIONS

We wish to calculate the sales tax on the 60 items, but only if the number of items sold is greater than zero. This example demonstrates the use of array references on both the right and left sides of an equation. It also shows how to create new variables with an array (the tax variables). These will be new numeric variables.

```
Array nitem [60] nitem1 – nitem60;
Array price [60] price1 – price60;
Array tax [60] tax1 – tax60;
```

```
Do I = 1 to 60;
        If nitem[i] gt 0 then tax[i] =
                nitem[i] * price[i] * .06 ;
End;
```

ARRAY FUNCTIONS

Let's review three SAS functions that pertain specifically to arrays:

Function	Definition
Dim(*arrayname*)	Returns the number of array elements
Lbound(*arrayname*)	Returns the lowest subscript
Hbound(*arrayname*)	Returns the highest subscript

You can find out the number of elements in an array by using the dim function. This is particularly useful when you ask SAS to count the number of elements for you by using the ' ***** ' in the array statement and you may not know how many elements are in that array. The dim function can be used as the ending number in a do loop. This enables you to create an array and run code through every member without ever knowing specifically how many members there are.

```
Array  b  {*}  b1-b14;
Do i = 1 to dim(b);

     Note:  dim(b) will return 14;
```

In addition to the DIM function, there are also the LBound and HBound functions. These are particularly useful if you specify your own subscripts. In the next example we are creating an array called items with subscripts from 23 to 25. The LBound of the array is 23 and the HBound of the array is 25. We can see the difference between Dim and HBound. If SAS uses its default numbering system, Dim() and Hbound() return the same number, unlike if you specify your own subscripts.

```
Array  items  {23:25}  var1 var2 var3;
Do i = LBound(items) to HBound(items);
```

```
The functions are given values as:
LBound (items) = 23
Hbound (items) = 25

Dim(items) = 3 (not used in example above)
```

EXAMPLE 3 – CLEANING VARIABLES

On many platforms, if the data are missing, the data field is assigned a 9 or 99 or 98, etc. This can play havoc with your results when calculating a mean or other summary statistics. The example below shows you one method of 'cleaning' all the numeric fields of these 'fake' missing values.

Note the following:

- The number of array elements is not specified. We are letting SAS count for us.
- The variable names are not specified. Instead, we are asking SAS to put all numeric variables in this dataset into this array.
- We do not explicitly know how many variables are going to be in this array. We can use the DIM function that returns the number of elements in an array and use it as the ending number for our do loop. This makes the program much more flexible and easy to use in other programs without modification.

```
Data one;
    set one;
    array nvar  {*}  _numeric_;
    do i = 1  to  dim(nvar);
            if nvar{i} in  (98  99  0)  then nvar{i} = .  ;
    end;
Run;
```

The following example loads character variables to the array and then uses a do loop to 'clean' them. Note the use of the LBound and HBound functions. In this case, the HBound and the Dim functions would return the same number. The LBound function returns a one.

```
    Data one;
            set one;
            array cvar  {*}  _character_;
            do i = lbound(cvar)  to  hbound(cvar);
                    if cvar{i} eq  '9'  then cvar{i} = '  ';
            end;
    Run;
```

EXAMPLE 4 – PROCESSING MULTIPLE ARRAYS IN ONE DO LOOP.

Here is another example. We have six tests, and each is made up of four sections. For example, Test 1 has sections Verbal, Math, Science, and History. Test 2 also has the same four sections, but different questions. This helps prevent cheating. For each of the 24 sections, we need to 'clean' the data; if a score greater than 80 is found (usually a 99 meaning student did not take this section), the score is re-set to missing.

```
    array verb {6};
    array math {6};
    array sci {6};
    array hist {6};
    do i=1 to 6;
            if verb{i}  > 80  then  verb{i} = .  ;
            if math{i}  >  80  then  math{i} = .  ;
            if sci{i}  >  80  then  sci{i} = .  ;
            if hist{i}  > 80 then  hist{i} = .  ;
    end;
```

TWO DIMENSIONAL ARRAYS

The previous example (4) can also be done using a two dimensional array. When defining a two-dimensional array, there are two numbers in the brackets separated by a comma. The first number is the number of rows and the second number is the number of columns. Sound familiar? Well yes, whenever you want to insert a table into Word, Word asks you those very same questions.

```
Array new  { # rows , # cols } varname1-varname?;
```

The second thing you need to know is that when you are loading variables into a 2D array, they are assigned a spot starting on row one and filling in across that row first (just like you are reading), before going to the next row.

So, 1) think of a word table, and 2) think of loading an array as if you were reading a book.

EXAMPLE 5 – REDO EXAMPLE 4 WITH A 2D ARRAY

Let's redo the array example from Example 4 using one 2D array. Note that we want four rows and 6 columns to correspond to the four sections and the six tests..

```
ARRAY   myArray  {4 , 6}
          verb1 – verb6
          math1 – math6
          hist1 – hist6
          sci1 – sci6;
```

This will load the array in the following way. Note how it is assigned left to right, top to bottom as you would read a book.

verb1	verb2	verb3	verb4	verb5	verb6
math1	math2	math3	math4	math5	math6
hist1	hist2	hist3	hist4	hist5	hist6
sci1	sci2	sci3	sci4	sci5	sci6

To refer to an element, you always specify the row you want first, then the column. Separate the two numbers with a comma.

myArray [3,2]	will return the value of the variable Hist2

A do loop is needed for each dimension, thus two do loops, are specified below, one for the rows (which is represented by i and set from 1 to 4) and one for the columns (represented by j and set from 1 to 6). Be careful that if you make i reference the rows (1 to 4), that i is put in the first position in the array reference.

```
Do i = 1 to 4;     * row;
        do j = 1 to 6;      * column ;
                if myArray[i , j]  >  80  then  myArray[i , j] = . ;
        end;
End;
```

TEMPORARY ARRAYS

Temporary arrays are the type of arrays generally found in other programming languages. There are no variables saved in the array, only the values, and the values are thrown away at the end of the data step.

A few things to keep in mind:

- You cannot load already existing variables into a temporary array
- You cannot create variables with a temporary array.
- The values that are loaded into a temporary array are not kept at the end of the data step.
- Values are retained across all observations (just as in other SAS initialized arrays)
- You cannot use a '*' to define or reference elements in a temporary array.

There are some benefits to using temporary arrays.

- They save a lot of space because the values are not kept for every observation.
- They process faster than using regular arrays with variables.

Temporary arrays are useful if you need the values only for a calculation within the data step.

EXAMPLE 6 – SCORING A TEST WITH 50 QUESTIONS

You are given the task of scoring an exam and computing how many questions each student answered correctly. The test has 50 questions 'items'. The correct answer to the test questions is the 'Key'. We can place the keys to all the test questions in a temporary array and use DO loop processing to score the answers.

```
Array  Raw  {*}  item1-item50;
Array  Key   {50} $  _temporary_
                ('B'  'C'  'A'  'B'  'D'  etc… etc…);
Array  Score  {50} ;

Do i = 1 to 50;
        if raw{i} eq key{i} then score{i}=1;
        else score{i}=0;
End;
TotalCorrect = sum( of score1 – score50 );
```

The first array loads the already existing test answers, item1 – item50. Since they previously existed as character data step variables, the $ in the array statement is not needed. Note: we are letting SAS determine the number of elements.

The second array holds the answer keys. Since we do not need to keep the key values as part of the dataset (they will not be used in any further processing), we can use a temporary array.

The third array is the score array. This will be used to indicate whether the student answered the question correctly (assigned a 1) or not (assigned a 0). Since these variables are new to the data step, this array statement CREATES the 50 variables called score1 to score50. Since we did not include a $ sign and these are new variables, SAS defines them as numeric.

Then the DO loop loops through all 50 questions, comparing the answers to the key and setting the scores appropriately. The last statement uses a SAS function to sum up the score variables to get a total score for each student.

USING ARRAYS INSIDE FUNCTIONS

In the last example, we saw that the TotalCorrect variable was created by a sum of the score variables. You can also use array references inside the sum statement. Below are one array and two summary statements. Both summary statements are equivalent.

```
Array  a  {4}  a1-a4;

Total = sum (a{1}, a{2}, a{3}, a{4});
Total = sum (of  a{*} );
```

In Example 6 above, TotalCorrect could be specified as either of the following (both are equivalent).

```
TotalCorrect = sum (of score1 – score50 );
TotalCorrect = sum (of score[*] );
```

One word of caution. The following will **NOT** work:

```
TotalCorrect = sum (of score[1] – score[50] );
```

Currently SAS does not allow the use of array references to define the starting and ending values of a range. SAS interpret the array references first (returning the appropriate values) before it does the sum function, so the last TotalCorrect calculation would substract score50 from score1 and sum the result of that calculation.

EXAMPLE 7: USING ARRAYS TO CALCULATE TOTAL ESSAY SCORES

In the following example a test of 20 essay questions will be scored. To determine the score two readers independently score each essay. The score for each essay is the average of the two reader's scores, rounded to an integer.

```
Array  reader1  {20}  rdr1q1 – rdr1q20;
Array  reader2  {20}  rdr2q1 – rdr2q20;
Array total  {20}  totalq1 – totalq20;

Do i = 1 to 20;
        total{i} = round(mean (reader1{i}, reader2{i} ), 1);
End;
TotalScore = sum (of total [*] ) ;
```

Three arrays are needed to calculate the scores, one for each of the two reader's scores, and one for the total score for each essay. The do loop then cycles through each of the 20 essay questions and calculates a total score. The last calculation sums the 20 essay scores to get a total for the student's test.

Note: we are using three functions here, the mean function, the round function, and the sum function. The mean function can have an unlimited number of parameters, and it returns the average of all of them. The round function requires two parameters, the first is the number you want to round and the second specifies the decimal level that you want to round to (or an integer to round to – see SAS documentation for more on this feature).

EXAMPLE 8: ZERO-FILL A CHARACTER FIELD

This example demonstrates the use of arrays and functions to right justify, then zero-fill multiple character fields.

```
array var{*} var1 - var50;
do i=1 to dim(var);
        if var(i) ne '  '  then var{i} = translate (right(var[i]),'0',‘  ');
end;
```

This example can be modified to accept any number of variables. The RIGHT and the TRANSLATE functions right justify and change the blanks to zeros, respectfully.

EXAMPLE 9: COUNTING ACROSS VARIABLES

In this example we will use the array function to check the number of schools a student has indicated on a form. The form allows the entry of up to ten school numbers. The function will count how many schools each student entered. The ten fields will either have a school identifying code or be blank. In this example, assume that once a blank field is found, all further fields are also blank.

```
count=0;
array school {10}  school1-school10;
do i = 1 to 10 while (school{i} ne ‘  ');
        count+1;
end;
```

First we set the count variable to '0' at the start of each observation. Then we set up an array with the 10 school variables. The do loop is an incremental loop combined with a do while loop. It will loop through each of the school fields in the array and increment the count by 1 if it finds a non-blank value. Once it finds a blank field, the loop is exited. Note this will not return the correct count if there are blank fields embedded in the middle of the ten fields.

USING ARRAYS TO WORK WITH LONGITUDINAL DATA (OR DATA ACROSS TIME)

In the following examples we will calculate summary statistics of the scores of the tests an individual student took over time. When the data are formatted in a way that does not allow correct calculation of the statistics, arrays can help. Using arrays, you can make several observations from one, or one observation from many. Here are two examples.

EXAMPLE 10: CREATING MULTIPLE OBSERVATIONS FROM ONE

Below is an example of some data. To calculate summary statistics across all tests for all the scores in the following data, we need each score as its own observation.

Examinee	Test1	Test2	Test3
Peter	150	200	175
Sara	320		360
etc.			

Here is the program that allows us to get the data in a form that we can use:

```
keep examinee score;
array sc {3} testscore1-testscore3;
do j = 1 to 3;
        if sc{j} ne '  ' then do;
                score = sc{j};
                output;
        end;
end;
```

Since we really only need two variables, we use a keep statement to save memory space in our final dataset. Our array has three entries, the three test scores for each student. The do loop processes through them, checks to see if there is a score in that field and if so, assigns the score to the 'score' variable and outputs an observation.

Note: we used an array name of sc. We could not use an array name of score because we want that to be the name of the dataset variable we are keeping. Arrays cannot have the same name as variables in the dataset.

Here is the output data created from the first two records listed above.

Examinee	Score
Peter	150
Peter	200
Peter	175
Sara	320
Sara	360

EXAMPLE 11: CREATING A SINGLE OBSERVATION FROM MANY

In the following example, you are asked to analyze the scores each student received over a period of several years, but each score is on a separate record. One way to analyze this is to use arrays to make a single observation for each student.

In this example, assume each student can have a maximum of five scores across five years. Note: This is easily customizable for more than five values. Here is a sample of the data:

Examinee	Year	Score
Peter	2000	250
Peter	2002	310
Peter	2001	280
Sara	2002	350
Sara	2004	400

In order to do this, we will make use of the first. and last. variables. The first.X and last.X variables are binary/logical variables (1/0) and their values are either a 0 (for false) or 1 (for true). You create these variables by using a SET with a BY statement inside a data step. Note, since we are using a BY statement, we must also sort by these same variables before the data step or SAS will return an error. First.X is assigned a '1' if this is the first observation for that value. Otherwise it is set to a zero. Last.X is assigned a '1' when this is the last observation the value appears. These variables are temporary and are not kept in the dataset after the data step finishes.

```
Proc sort data=one;
        by examinee;
Run;
Data one;
        array scores [2000:2004] score2000 – score2004;
        do until (last.examinee);
                set one;
                by examinee;
                scores[year] = score;
        end;
Run;
```

The first step is to sort the data by examinee ID. Then inside the data step, we first create an array that will hold the five scores. Note the index of the array values are not 1 to 5, but the year. Then we set up a loop that will continue to set observations from dataset one to the current observation using the year variable as the array index. This continues until we reach the last observation for each student at which time the loop is exited and the observation is output to the dataset (implicitly).

Examinee	Score2000	Score2001	Score2002	Score2003	Score2004
Peter	250	280	310		
Sara			350		400

Note: This example could also have been accomplished using the Transpose procedure. The following code will produce the same output given the input above.

```
Proc transpose data=one out=newds prefix=score;
        BY name;
        ID year;  *names the variables in the new dataset along with the prefix option.;
Run;
```

EXAMPLE 12: READING VARIABLE LENGTH DATA RECORDS

The last example is the most complicated. Assume you have data that have a variable number of blocks with data in each. This can successfully be read into a dataset using arrays.

For example, assume you have a layout that looks like this:

- 1-50 fixed length all vars in same cols
- 51-52 number of score info sections (up to 12)
- 53-54 number of institution codes to send scores to (up to 15) – called (DI's)

  Note: DI = Designated Institution to receive scores

- Followed by:
  - Up to 12 sets of scores – 25 chars each.
  - Up to 15 DI codes – 4 chars each. This section starts at the end of score section (no spaces between the two sections).

The first 50 columns always hold the same variables and do not change. The next two fields tell us how many score sections there are on this record and how many DI sections there are. Next on the record is either the first score section, or, if there are no score sections, then the first DI section. Note the score sections are always a length of 25 and the DI sections are always a length of 4.

- Example of Records in the above layout.

Columns 1-50	Columns 51-52	Columns 53-54	rest of record
Name1 ssn1 address1	2	3	score1score2di1di2di3
Name2 ssn2 address2	1	4	score1di1di2di3di4
Name3 ssn3 address3	3	0	score1score2score3
Name4 ssn4 address4	0	2	di1di2

The code to read this data file is:

```
input    @1   fixed   $char50.
         @51 nscores   2.
         @53 ndis   2.
         @;

array score {12} $25 score1-score12;
array di {15} $4 di1-di15;

if nscores gt 0 then do i = 1 to nscores;
         input score{i} $char25. @;
         verbal = substr( score{i}, 15, 2);
         math = substr(score {i} , 17, 2);
end;

if ndis gt 0 then do i = 1 to ndis;
         input di{i} $char4. @;
end;
```

First we read in the first block of data and the two counts of the number of score sections and the number of DI sections. Note the '@' sign at the end of the input statement. This holds the cursor at the same record and actually even at the same spot on the record. So at this point, it is sitting at position 55 (after it read in the 2 positions at 53 and 54 for 'ndis').

We need to set up two arrays, one for the score sections and one for the DI sections. For the score array, there can be up to 12 score sections so that is the number of array elements we need. Each is a length of 25 so that is the length used. For the DI array, there can be up to 15 of those, with each being a length of 4. Both score and DI are character arrays.

The first IF statement checks to see if there are any score sections to read in. If so, then a loop is executed to read in each of the 25 character chunks. Note: You will still need the @ sign at the end of the input statement so that the next input command does not start on the next line. Read the score data in chunks of 25 chars each. This sets the cursor at the correct spot to read in the next chunk. Then use the function substr to parcel out the pieces of data you need out of the chunk. Here we are substringing out the verbal and math scores.

The next IF statement checks to see if there are any DI chunks. If so, then a loop is executed to read in each of them.

Since the variables in these arrays are not initialized, they are not retained. Otherwise we would have needed a separate do loop to clear out the values from the last observation.

CONCLUSION

I hope the techniques provided here will prove useful to you in the future. These topics are just a few examples of the power of arrays. They have certainly made my life much easier (a LOT less typing!).

SAS and SAS/GRAPH are registered trademarks or trademarks of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

AUTHOR CONTACT

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

Wendi L. Wright
11-L Educational Testing Service
Princeton, NJ 08541
Phone: (609) 683-2292
Fax: (609) 683-2130
E-mail: wwright@ets.org