Paper 179-2008

Using Do Statements, Links, and Arrays

Jimmy DeFoor
Program manager, Data Base Marketing
Citi Card, Irving, Texas

Abstract

This paper will cover the basics of Do statements, links, and arrays, while also describing the use of the SET POINT option to access a SAS data set by specific line number. In addition, it will show using

- Do Loops and Arrays to pull descriptions or labels.
- Do Loops and the SET POINT option to randomly select a specific number of records from a SAS data set.
- Do Loops and the SET POINT option to retrieve only the records that match only a specific condition, such as outcome to a particular drug trial for a specific group of patients.
- Returns to determine when processing moves back to the top of the DATA Step ®.

Introduction

Do statements, links, and arrays are part of the tool set of the good SAS programmer. They not only increase the functionality of the SAS DATA Step, they also enhance program clarity and control while reducing maintenance effort.

Do Statements

The Do statement is the primary control block of DATA Step coding. It answers the essential questions of when and how often a block of code is to be executed. Here's the simple syntax of the Do statement, with descriptions in parenthesis.

Do; (Evaluate whether code is to be executed.)

Code block to execute.

End; (Make decision whether to exit code or return to Do.)

Of course no one uses the simple Do syntax because it would just add more coding without adding more functionality. Still, it is a valid coding form. Try it and you will see that the DATA Step compiler had no problem handling it.

Often the Do statement is found as the action component of an If statement.

```
If Day = 'Monday' then
  Do;
     Weekspay = 0;
      Weekstax = 0;
  End;
```

```
Array week(7) day1 – day7;
If Day = 'Monday' then
  Do j = 1 to 7;
     Week(j) = 0;
  End;
```

Most coders probably don't think of the first form as a Do group structure because it doesn't have a looping construct; but it is just as much a Do group as the second form, which is the an iterative Do group with index variable.

Iterative Do Group
With Index Variable

```
X = 0;
Do j = 1 to 10;
    X = X + 1;
End;
```

The basic Do statement has the same decision points as an iterative Do group. 1) Decide whether the code should be executed. 2) Decide whether the action should return to the beginning of the Do group. For the basic Do statement, obviously, the decision to return is always 'No'.

Iterative Do groups loop through a body of code until stopped by the control variable. There are three types of Do groups that are Iterative: Index Variable, Do Until and Do While. The Index Variable group just adds the useful feature of automatically incrementing the control variable each time it passes through the loop. The Do While and Do Until groups require that the programmer add code to increment or reset the control variable. This can be seen in the examples below, where J is the control variable.

Do Until

```
X = 0;
J = 1;
Do until (J ge 10);
    X = X + 1;
    J = J + 1;
End;
```

Do While

```
X = 0;
J = 1;
Do while (J le 10);
    X = X + 1;
    J = J + 1;
End;
```

What's the difference between Do Until and Do While groups? The typical description is:

- Do While statements evaluate at the top of the loop.
- Do Until statements evaluate at the bottom of the loop.

Although not completely accurate, since both forms must go through the evaluation in the End construct before actually exiting the Do group, this explanation does effectively communicate the important difference in the two structures. Do While statements do not start before a particular value exists in the control variable; Do Until statements begin without evaluation and execute until control variable has a particular value.

This difference can seen be seen by modifying both sets of code so that J starts at 11 instead of 1. The Do Until structure will still execute, which means that X and J will both be incremented by one. The Do While structure will not execute, however, so both X and J will continue to have their initial values.

Do Until

```
X = 0;
J = 11;
Do until (J ge 10);
    X = X + 1;
    J = J + 1;
End;
```

Do While

```
X = 0;
J = 11;
Do while (J le 10);
    X = X + 1;
    J = J + 1;
End;
```

This distinction shows the control problem of the Index Variable and Do Until groups. They can create unwanted results unless they are controlled by an additional If statement. Do While statements are safer structures because an initial evaluation is an inherent part of their construct. They will never execute unless the conditions are as expected. This restriction can be especially important if the Do group is executed via a macro call, a link, or a %include because its application will be more universal and, hence, more likely to encounter opportunities for misuse.

An unadvertised form of the iterative Do group is the Do Over. It processes arrays just like the Index Variable group, but without having to specify the Index Variable or its control value. The total number of elements in the array is passed from the DATA Step to the Do Over. The coding forms below access their arrays exactly the same way: from element one (1) through element five (5).

Do Over

```
Array Test X1-X5 (1,3,5,7,9);
Do over test;
   Put test(_I_)=;
End;
```

Index Variable

```
Array Test (5) X1-X5 (1,3,5,7,9);
Do J = 1 to 5;
   Put test(j)=;
End;
```

Both of these examples use the initial values option to create five variables, X1 – X5, and load numeric values to those variables. Character variables could have been created by placing a $ in front of the variable names in the array and by putting each of the values in quotes. More will be discussed about arrays later in this paper.

The Do Over syntax is no longer documented by SAS, but it is still available since it was released in earlier versions of Base SAS. Info on its usage can be found in the SAS Global Forum proceedings at http://support.sas.com/events/sasglobalforum/previous/online.html. Just be aware that it can only be used with implicitly subscripted arrays, which is why Test doesn't have a subscript in that example. Also, implicit arrays have the step-generated counter of _I_.

The benefit of the Do Over form can now be obtained with the Dim function, which returns the number of elements in an array. The '`Do Over Test;`' shown above would be changed to '`Do J = 1 to Dim(test);`' and the '`Put test(_I_)=;`' would become '`Put test(j)=;`'.

All four of the forms of the Do Statement have their particular usefulness, especially for program clarity; but the function of any of them can be executed by the Do While statement.

The Do While form also most approximates the looping action of the SAS DATA Step in the processing of a group of records. To see this, create a small data set and then read it using the typical DATA Step on the left. The DATA Step stops just before reading Work1 again *after* the last record has been processed. Thus, the DATA Step stops at the top of its loop through the records of Work1 and then exits the Step. This can be seen in the log from the order of the output of the Put statements: B follows C. If the data loop stopped at the end of the Step, the order would be A, then C, and there wouldn't be a B.

Typical SAS DATA Step

```
Data _Null_;
  If _n_ le 1 then
    put 'A ';
  If eof then
    put 'B ';
  Set work1 end=eof;
  If eof then
    put 'C ';
Run;
```

Order of Values in SAS Log

```
A
C
B
```

Using a Do While to Read a SAS DATA Step with the SET POINT = Option

```
Data work2;
  Readrow = 1;
  Rowobs  = 1;
  Do while (readrow);
    If rowobs gt rowtot then
      readrow = 0;
    Else
      do;
        set work1 nobs=rowtot
           point=rowobs;
        Rowobs = rowobs + 1;
        Output;
      end;
  End;
  Stop;
Run;
```

Coding the read of a SAS data set within a Do Loop is not just an intellectual exercise to demonstrate the typical looping process in the DATA Step. It is also the only way that a SAS data set can be read by specific observation, which is done with the SET POINT= option.

The SET POINT option enables the SAS program to move up or down a SAS data set according to the line number passed to the Point variable, which is rowobs in this example. This option will be discussed more later, but note for now that the rowobs variable isn't in a Retain statement and yet its values are still being retained during each pass through the Do loop. This is because reading a Data Set in a Do loop prevents variables from being automatically reset to blank or missing.

Return Statement

The iterative Do group obviously has a built-in return feature. Execution returns to the top of the loop until a control value terminates the loop. The typical SAS DATA Step also has such a built-in return; otherwise, it would not process another record from the source file. But a return in a DATA Step can be forced before the default return by adding a Return statement anywhere in the DATA Step. And the Return statement can be stand-alone or part of an evaluation, such as

```
If City ne 'Fort Worth' then return;
```

The use of a 'non-linked' Return statement anywhere in a DATA Step will alter the location of the implicit output statement that is part of every DATA Step that doesn't have an explicit output statement. The DATA Step will place the implicit output statement *above* the return and not at the end of the DATA Step, as it does normally. Thus, a Return statement does not act the same as a Delete statement, which will always have the implicit output placed after it.

```
If City ne 'Fort Worth' then delete;
```

Of course, if it did, then it wouldn't be of much value.

Return statements are not particularly useful unless they are combined with Link statements. There, they have two functions:

1. Return the execution of all unlinked processing to the top of the DATA Step.
2. Return all linked processing to the line where the link was first referenced.

Link Statements

A Typical Link Statement

```
If index(field,'street') ne 0 then
  Link address;

-- code for all records ---

Output;
Return;  /* return to top of step */
Address:

-- Code for address records only --

Return; /* return to link */
Run;
```

Link statements are used to separate code from the normal flow of the DATA Step in order that the code can be executed from any location in the Step. This prevents the need to place the same code in multiple locations in the Step, which greatly reduces the labor to modify the Step when the code must be changed.

Some of the best coding candidates for use in links are common editing routines for multiple address fields or computations that apply to multiple numeric fields.

In the code above, execution skips to the line just below the statement label 'Address:' whenever the variable field contains 'street'. All other records continue processing through the code that leads to the Output statement. At that point, the first Return shifts execution back to the top of the DATA Step. On the other hand, the address records that jumped to the code that is below 'Address:' will move to the second Return, which takes them back to just below 'Link address'. They then move through the same code as the other records before returning to the top of the Step. All records pass through the explicit Output statement above the first Return, which is the location where the implicit Output would have been placed had it not been coded.

Clearly, this example could just as easily be executed in a Do statement, so using a Link statement here is of no value unless there are other potential address fields that must be evaluated and edited in the DATA Step. That is the key. Links are very useful, but only if the same processing is applicable to multiple fields.

Arrays

Arrays are nothing more than groupings of variables. They are not permanent data structures. They exist only for the duration of the DATA Step. The names given arrays are completely independent of the variable names in the grouping, but it is usually wise to employ names that are related to the content of the variables in the array. For example, the Array Monthly would be a sensible grouping of the M1 through M12 if the variables contained monthly values from January through December.

Array Monthly (1:12) M1 – M12;

Array	1	2	3	4	5	6	7	8	9	10	11	12	elements
Monthly	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	variables

The individual components of an array are called elements or references. Their order matches the order of the variable specification: the first variable is in the first element; the second variable is in the second element, and so on. Hence, the elements are typically numbered from one (1) to the maximum number in the grouping. For that reason, the DATA Step defaults to one (1) as the lower bound when a bound isn't stated. Thus, the usual coding for the above array is 'Array Monthly (12) M1 – M12', but the structure is really 'Array Monthly (1:12)   M1 – M12'. Starting at 1 isn't required, however. The elements could have started at any number, such as 'Array (0:11)' or 'Array (61:72)', as long as the upper bound was greater than the lower bound and the number of elements in the array matched the number of variables in the array; e.g., 'Monthly(0:11) M1-M12'.

This flexibility, which is especially needed when new array elements can be added frequently to existing elements, provides one reason for the use of the Lbound and Rbound functions. They return the lower and upper bounds of an array. Their use eliminates the need to recode the Index Variable Do groups that address those arrays when the dimensions of the arrays change size.

Lbound and Hbound functions

```
Array sales  (1:60) s1  -  s60;
Array costs  (1:60) c1  -  c60;
Array csratio(1:60) cs1 - cs60;
Do j = lbound(sales) to
       hbound(sales);
  Csratio(j)= cost(j)/sales(j);
End;
```

In this code there are three arrays of the same dimension. When those dimensions change, the Array statements have to be modified, but the Do Loops that access the arrays don't need to be recoded if they use the lbound and hbound functions. When there are multiple Do Loops for the same arrays, this can be a major time-saver.

Though arrays are usually groups of variables with similar content, such as hourly temperature readings or patient test results, the variables don't need to have any common qualities except that they are to be acted upon in the same way. Suppose a group of character variables named Z, T20, Daylast, NT, and B12 are to be written to a delimited file and it is desired to allocate the smallest necessary space for the file.

Calculating the needed file size requires determining the largest-sized record that must be written. This requires knowing the maximum number of characters in each field, summing those values, and then adding the number of bytes needed for the delimiters. Processing the variables as an array within an Index Variable Do group finds that value.

Creating an Array of Known Character Variables

In this example, the enclosed asterisk (*) syntax is used in the array statement so that the DATA Step will calculate the total number of fields in the array. Next, the Dim function is used to set the end point of the looping process. This Dim function was shown earlier in the Do Statements section of the paper. As stated there, the Do Over method could also have been used here instead of the Dim function, just as could the Lbound and Hbound functions.

```
Set charvars;
Retain max_length 0;
Array charvars (*) Z T20 Daylast
                   Nt B12;
Nonblank = 0;
Do J = 1 to dim(charvars);
  Nonblank = length(charvars(j)) +
             Nonblank;
End;
Total_length = Nonblank +
               dim(charvars) - 1;
If total_length gt max_length then
  max_length = total_length;
```

In the above example, the character variables were known and were the only variables on the data set. But what if they weren't known, or what if there were also numeric variables, how could of the maximum length of the record been calculated in that case?

Using the _Numeric_, _Character_, and _All_ Variable Lists in Arrays

The _All_ list references every variable in a SAS data set, but that is of little value in arrays because the variables in an array must either be all character or all numeric. They cannot be both. Hence, the _All_ list can be used only if variables are known to be one or the other. In that case, it would be simpler and clearer in purpose to use only the _Character_ or _Numeric_ list that applied to the variables in the data set. When the formats of the variables in data set can be either character or numeric, two arrays – one of each type - must be used to calculate the maximum length of the records to be written to a delimited file. This approach is shown below.

```
Data Max(keep=max_length);
 Set allvars;
 Retain max_length 0
        Onechar   ‘ ‘;
 Array charvars (*) _character_;
 Array numvars  (*) _numeric_;
 Nonblank = 0;
 Do J = 1 to dim(charvars);
   Nonblank = length(charvars(j)) + Nonblank;
 End;
 Do J = 1 to dim(numvars);
   Nonblank = length(left(put(numvars(j),12.0))) + Nonblank;
 End;
 Total_length = Nonblank + dim(charvars) + dim(numvars) – 1;
 If total_length gt max_length then
   max_length = total_length;
 if eof then
   output;
Run;
```

Those who read the code carefully will understand that it over-calculates the maximum record length because it includes temporary variables such as nonblank that will not be written to the delimited data set. They will also see that it creates a character variable, onechar, that appears to have no use in the DATA Step. Yet, it does. Its presence assures that the charvars array will be defined, just as max_length variable assures that the numvars array will be defined. In this way, the DATA Step will work regardless of the variable types in allvars. Otherwise, the step could fail because of ‘zero elements’ in one of the arrays.

Accessing Array Elements and Creating Their Content

Arrays of permanent variables can be referenced as individual elements or as variables. The code of ‘`Monthly(12) = 15`’ produces the same result as ‘`M12 = 15`’.

Arrays can be accessed forward or backward. In this example, which reverses the content of the array elements, the array is accessed both forward and backward via K and J, respectively. This was done so that further array processing in the DATA Step could be from 1 to 12.

```
Array Monthly (1:12) M1 – M12;
K = 1;
Do J = 12 To 7 by -1;
  Tempvar    = Monthly(K);
  Monthly(K) = Monthly(J);
  Monthly(J) = Tempvar;
  K = K + 1;
End;
```

Array elements can also be skipped or accessed by specific position.

`Array Sales (1:60) s1-s60;`
`Do J = 1 to 30 by 3;`
	`Array Costs (1:60) c1-c60;`
	`Do J = 1, 9, 5, 15, 45, 50 to 56;`

The contents of array elements are usually provided from existing variables.

`Set Salesrec;`
`Array Sales (1:60)  s1-s60;`
	`Retain cr1-cr60 0;`
	`Array Csratio(1:60) cr1-cr60;`

But they can also be loaded directly into the variables from the Array statement.

```
Array Codes  (1:8)   4 (1, 3, 5, 6, 8, 10, 20, 35);

Array City   (1:6) $12 (‘Fort Worth’, ‘Dallas’, ‘Arlington’
                        ‘Irving’, ‘Euless’, ‘Haltom City’);
```

The bounds must be stated explicitly when the initial-value technique is used to define variable values. Also, formats must be specified for character data and are advised for numeric. Here the numeric length is set to four (4) to reduce storage space, while $12 is used to prevent truncation of any of the character values.

This initial-value technique causes the DATA Step to create permanent variables with the array name as a prefix. For example, the variables Code1 through Code8 are created in the first example and City1 through City6 are created in the second. To prevent the creation of permanent variables, add the keyword ‘_temporary_’ just before the list of initial values.

```
Array Codes (1:8) 4 _temporary_ (1, 3, 5, 6, 8, 10, 20, 35);
```

Using Arrays instead of User Formats to Assign Descriptions

Often data contains numeric codes for which the descriptions of those codes are stored elsewhere. These descriptions could be loaded into a user format and then applied to the data with a Put function such as ‘`Description = Put(code,$descrptn.0)`’. This is an excellent technique and can be implemented easily with the CNTLIN option to build a user format from a SAS data set. An alternative technique would be to load the descriptions into an array and then add those descriptions to each row in the data using a Do Loop. This approach saves processing expense because arrays can process much faster than user formats and the use of an array for table lookup is not difficult.

The following program reads the description data from a SAS data set, loads it into an array using element values that are the same as the description codes, and then applies those descriptions to the source data based upon those codes. The first step uses two Call Symput statements to create two macro variables: one to hold the maximum length of the descriptions, the other to hold the largest-valued code associated with the descriptions. The codes and descriptions do not have to be unique, but only the last description of a code will be kept if there are multiple entries.

```
Data _null_;
  Retain maxcode 0
         maxlgth 0;
  Set Descriptions end=eof;
  If length(description) gt maxlgth then
    maxlgth = length(description);
  If code gt maxcode then
    maxcode = code;
  If eof then
    do;
       call symput('maxlgth',left(put(maxlgth,8.0)));
       call symput('maxcode',left(put(maxcode,8.0)));
    end;
Run;
*;
Data newdata (keep=code description);
  Length description $&maxlgth;
  Length description1 - description&maxcode $&maxlgth;
  Retain description1 - description&maxcode ' ';
  /* read description file and build array by code */
  Do while (not eof1);
    Set descriptions end=eof1;
    Array descrptn (1: &maxcode) description1 - description&maxcode;
    Descrptn(code) = description;
  end;
  /* read code file and retrieve descriptions by code */
  Do while (not eof2);
    Set srcdata end=eof2;
    Description = descrptn(code);
    Output;
  End;
  stop;
Run;
```

The second DATA step uses the macro variables to set the number of description variables, the length of those variables, and the number of elements in the Descrptn array. Next, it loads the descriptions into the array so that each array position corresponds to the description code. Then, at the end of the description file, it reads the source data, pulls the descriptions from the array by the code on each record, and loads them into the Description field. The reading of each data set is controlled by a Do While statement that reads while not at end-of-file.

This technique can only be used with numeric codes because array elements in SAS can only be numeric. Also, the technique builds an array with a lower bound of 1 and an upper bound equal to the largest code in the array. This means that the array will have blank descriptions for each code that isn't defined. This won't affect the effectiveness of the technique, however, unless the Description data set doesn't have all of the codes that are in the source data set. However, a user format built from the same data set would also have that shortcoming.

Using Do Loops and Arrays to Directly Access a SAS Data Set

As discussed earlier in the paper, the SET POINT= option accesses a SAS data set directly by line number. SAS calculates the offset from the current position in the SAS data set to the line number specified in the Point= variable and then pulls the record associated with that line. If only a small percentage of the records must be retrieved from a data set, then the SET POINT option saves a lot of unnecessary I/O. This is the opportunity presented by simple random sampling. Furthermore, using the SET POINT option in random sampling is also one of the most efficient methods of getting a specific number of records from a SAS data set.

Simple Random Sampling without Replacement

```
Data sample(keep=record name age sex);
  Retain count 0       row1 - row10 0;
  Array  row (1:10) 4 row1 - row10;
  Count    = 1; Complete = 0;
  Do while (not complete);
    Match  = 0;
    Record = (round(uniform(2) * totobs));
    Do j = 1 to count;
      If record = row(j) then
        match = 1; /* Is record not new?*/
    End;
    If not match then
      do; /* add to array if new */
         row(count) = record;
         count      = count + 1;
      end;
    If count gt 10 then
      complete = 1;/* array now complete*/
  End;
  Do j = 1 to dim(row);
    record = row(j); /*
    set observs nobs=totobs point=record;
    output;
  End;
  Stop;
Run;
```

This code pulls a sample of 10 people. Line numbers are generated using the uniform function and the total number of observations in the data set. Each line number is loaded into an array unless it is already there. When the array has 10 unique line numbers, the sampling is ended. The array content is then loaded one element at a time into the variable ‘record’ and that variable is passed to the Point option so that the desired line can be retrieved and output to the data set Sample.

Each array element is loaded into the variable ‘record’ because the Point option will not accept array elements. A Stop statement is placed at the completion of the data retrieval because the Point option requires a Stop to end the DATA Step.

Another use of the SET POINT option is to retrieve only those records that match a particular characteristic, provided that those characteristics are tied to line numbers. For that situation to exist, a separate data set must be available that has the line number positions for each characteristic. Suppose there is a data set 'Id' that has a unique identifier for each subject in a drug review and that each row on that data set has the line number in which the various blood test results of that subject have been stored in another data set called 'Observs'. If that is the case, then the Id data set could be used to retrieve the test results from the Observs data set. The following code uses an array and the SET POINT option to retrieve those test results.

This code retrieves all the line numbers associated with the blood test results for a unique id and loads them into an array. It then retrieves those line numbers from the data set 'Observs' that contains the test results. The first.id and last.id variables are used to control the processing. They prevent having line numbers for more than one id loaded into the Row array. When last.id is true, records are read from data set Observs that correspond to the line numbers stored in the array for that Id. The DATA Step is stopped when all records in Observs have been read from the last id in the Id data set.

Again, as stated above, the Stop statement must be used in the DATA Step or the step will not terminate. This is a necessary condition of using the SET POINT option.

```
Data Work1(keep=Id Record Tr1-Tr50);
  Retain Count      0
         Tr1 - Tr50 0;
  Set id end=eof;
  Array row (1:50) 4 Tr1 - Tr50;
  By id;
  If first.id then
    do;  /* clear contents of array */
      count = 1;
      do j = 1 to 50;
        row(j) = .;
      end;
    end;
  Row(count) = line;
  If last.id then
    Do j = 1 To count;
      record = row(j);
      set observs point=record;
      output;
    end;
  Else
    Count = count + 1;
  If eof then
    stop;
Run;
```

A valid question, of course, is why would anyone ever need to write code like this? There are two reasons:

- Removing Ids from the Observs data sets prevent anyone from being able to tie the data to a particular individual or test group. This gives more security to the test participant and to the test process.
- The amount of data on each row in the Observs data set may be very large and most access of the data is either for particular ids or randomly selected ids.

Both of these objectives can be solved by putting the ids on a second data set that also has the line numbers on the Observs data set for the test results for each of those ids. Then, the data can be retrieved from Observs by particular Id or by random selection using the SET POINT= option.

If this is true, some will ask, why not use an index on the second data set? Wouldn't it allow the same functionality as the line numbers? It would, provided a proxy variable was created for each ID and the proxy variable was used to index the Observs data set. However, managing index lookups in the SAS Data Step for multiple retrievals on the same ID is much more challenging than using the SET POINT= option. Proc SQL ® is a better technique of handling one-to-many retrievals than the SAS DATA Step for indexed variables.

Also, implementing this retrieval method is not difficult. Below is some code that would add line numbers to the Id data set while also adding the test results to the Observs data set.

Creating Line Numbers for the Use of Direct Access Retrieval

```
Data Add_Ids (keep=line id);
  Set Observs nobs = totobs;
  Complete = 0;
  Line = totobs;
  Do while (not complete);
   set newobs(keep=id) end=eof;
   line = line + 1;
    output;
    if eof then
     complete = 1;
  end;
  stop;
Run;
*;
Proc datasets library = work;
  Append base=observs
         data=newobs force;
  Append base=id
         data=add_ids force;
run;
```

This code determines the total observations currently in the Observs data set from the NOBS option on the Set statement. It then uses that number as the baseline for the line numbers that will be added to the Id data set. To that number, the code adds one (1) every time it finds a new observation that will be written to the Observs data set.

Thereafter, the new test results are appended to the Observs data set and the new line numbers are added to the Id data set. The force option is used so that variables not on Observs and Id will be dropped. This prevents the Id variable in Newobs from being added to Observs. Without this option, the appends will fail whenever Newobs and Add_ids have additional variables.

The Stop was added to the DATA Step to terminate processing once Newobs had been read completely. There was no reason to process anymore records from Observs once the total observations in Observs had been determined.

Conclusion

This paper has attempted to provide useful information on Do groups, Arrays, and Links, while also giving some suggestions on possible applications of those techniques. It has also attempted give a little insight into the functioning of the SAS DATA Step. Only the reader can judge whether the effort was successful.

Trademark Citations

Acknowledgements

Many thanks go to Dianne Piaskoski for her proof-reading and her very helpful suggestions. Her contribution far exceeded the payment she received.

Contact Information

Jimmy DeFoor
Fort Worth, Texas
jimmy.a.defoor@citi.com

References: Just a Few of the Good Sources Among the Many

First, Steven and Teresa Schudrowitz, “Arrays Made Easy: An Introduction to Arrays and Array Processing”, *Proceedings of the SAS Users Group International 30, 2005, <http://www2.sas.com/proceedings/sugi30/242-30.pdf>*

Virgile, Bob, “Changing the Shape of Your Data: PROC TRANSPOSE vs. Arrays”, *Proceedings of the SAS Users Group International 24, 1999, <http://www2.sas.com/proceedings/sugi24/Begtutor/p60-24.pdf>*

Buck, Debbie, “A Hands-On Introduction to SAS® DATA Step Programming”, *Proceedings of the SAS Users Group International 30, 2005, <http://www2.sas.com/proceedings/sugi30/134-30.pdf>*

Winn, Tom, “Guidelines for Coding of SAS® Programs”, *Proceedings of the SAS Users Group International 29, 2004, <http://www2.sas.com/proceedings/sugi29/258-29.pdf>*

Howard, Neil, “DATA Step Essentials”, *Proceedings of the SAS Users Group International 27, 2002, <http://www2.sas.com/proceedings/sugi27/p050-27.pdf>*

DeFoor, Jimmy, “Selecting Records and Assigning Attributes”, *Proceedings of the SAS Users Group International 27, 2002, <http://www2.sas.com/proceedings/sugi29/261-29.pdf>*