PharmaSUG 2012 – Paper AD29

# Innovative Techniques: Doing More with Loops and Arrays

Arthur L. Carpenter
California Occidental Consultants, Anchorage, AK

## ABSTRACT

DO loops and ARRAY statements are common tools in the DATA step. Together they allow us to iteratively process large amounts of data with a minimum amount of code. You have used loops and arrays dozens of times, but do you use them effectively? Do you take advantage of their full potential? Do you understand what is really taking place when you include these statements in your program?

Through a series of examples let's look at some techniques that utilize DO loops and ARRAYs. As we discuss the techniques shown in the examples we will also examine the use of the loops and the arrays by highlighting some of the advanced capabilities of these statements. Included are examples of DO and ARRAY statement shortcuts and 'extras', the DOW loop, transposing data, processing across observations, key indexing, creating a stack, and others.

## KEYWORDS

DO loop, DO UNTIL, DOW loop, ARRAY statement, DIM function, SET statement options

## INTRODUCTION

Although most SAS DATA step programmers have made use of DO loops and arrays, few take full advantage of the power and flexibility of these tools. There are many variations of the DO statement and these can be used in conjunction with arrays to accomplish a number of tasks that can otherwise be quite intractable. The primary objective of this paper therefore is to demonstrate the interaction of DO loops and arrays.

In the examples that follow, I have for the most part, assumed that that the reader has at least a passing understanding of the:

- types of DO loops and their basic syntax
- ARRAY statement and its various forms

Many of the examples in this paper have been borrowed (with the author's permission) from the SAS Press book *Carpenter's Guide to Innovative SAS® Techniques* (Carpenter, 2012). When appropriate a reference to this book will be made in the form of "(3.1.2)". This indicates that additional discussion can be found in Chapter 3 Section 1.2 of that book.

### Shorthand Variable Naming (2.6.1)

Several shorthand naming conventions exist for specifying a list of variables.

A set of variables with a common prefix and a numeric suffix can be used wherever a list of variables can be specified. When used in an ARRAY statement, variables not already on the PDV will be created.

```
array vis {10} visit1 - visit10;
```

All variables with a common prefix regardless of the suffix can be addressed using a colon.

```
array vis {10} visit:;
```

Specialized name lists can be used to address variables by their type. Since each of these lists will pertain to the current list of variables, they will not create variables. In each case the resulting list of variables will be in the same order as they are on the Program Data Vector

- `_CHARACTER_` All non-temporary character variables
- `_NUMERIC_` All non-temporary numeric variables
- `_ALL_` All non-temporary variables on the PDV

```
array allnum {*} _numeric_;
```

## DO Loop Specifications (3.9.2)

The iterative DO loop specification is probably the one most commonly used. However there are a number of variations that are less commonly applied.

The iterative DO can accept compound loop specifications. Each specification is separated by a comma.

```
do count=1 to 3, 5 to 20 by 5, 26, 33; . . . end;
```

In this loop the value of COUNT takes on the values of: `1, 2, 3, 5, 10, 15, 20, 26, 33.`

The index variable can also be character with each individual value specified.

```
do month = 'Jan', 'Feb', 'Mar'; . . . end;
```

The iterative DO loop is evaluated at the bottom of the loop. This means that the index variable is incremented and then evaluated. For the following DO loop, the variable COUNT exits the loop with a value of COUNT=4.

```
do count=1 to 3; . . . end;
```

Sometimes we want to prevent the index variable from being incremented beyond the maximum value. When the following DO loop terminates the variable COUNT exits the loop with a value of COUNT=3.

```
do count=1 to 3 until(count=3); . . . end;
```

The DO UNTIL and DO WHILE loop forms of the DO loop will be executed indefinitely until some exit criteria is met. For these loops the index variable must be incremented manually by the programmer. It is also possible to set up an infinite loop with conditional exit and increment the index variable automatically. Notice that this iterative DO loop does not have a TO specified.

```
do k=1 by 1 until(x=5); . . . end;
```

## ARRAY Statement Forms

Although the simple use of the ARRAY statement is well known, it also supports a number of less commonly used alternate syntax structures.

The dimension of the array (number of elements is enclosed in braces). Here the dimension of the array is 3, and the index starts at 1.

```
array list {3} aa bb cc;
array list {1:3} aa bb cc;
```

Typically the index for the array starts with one and has a maximum value of the dimension. However the index can start at any numeric value. The following array also has a dimension of 3, however its index starts at 0, and can take on a maximum value of 2.

```
array list {0:2} aa bb cc;
```

When the number of array elements is unknown, an asterisk can be used to cause SAS to determine the dimension for you. In each of these arrays has a dimension of 16 and each addresses the same list of variables (VISIT1-VISIT16).

```
array vis {16} visit1-visit16;
array vis {*} visit1-visit16;
array visit {16} ;
```

When the number of variables is unknown you will have to let SAS determine the dimension.

```
array nvar {*} _numeric_;
array cvar {*} _character_;
```

Initial values can be inserted into an array through the use of parentheses that enclose a list of initial values. Here the character variables CLIST1 – CLIST3 will be initialized to ‘a’, ‘b’, and ‘c’ respectively.

```
array clist {4:6} $1 ('a', 'b', 'c');
```

## Temporary Arrays

Temporary arrays can be especially useful when you need the power of an array, but do not have or want the values to be stored in variables. A temporary array stores values on the PDV in temporary locations that are not transferred to the new data set. The keyword _TEMPORARY_ is used instead of a variable list, and the dimension must be specified.

```
array visdate {16} _temporary_;
```

Like other arrays it is possible load initial values. These two array specifications are the same.

```
array list {5} _temporary_ (11,12,13,14,15);
array list {5} _temporary_ (11:15);
```

In the array ALIST all six values are initialized to 3, while the initial values of BLIST are 1,2,3,1,2,3.

```
array alist {6} _temporary_ (6*3);
array blist {6} _temporary_ (2*1:3);
```

## Simple Example – Transposing Data

Most, but not all, SAS procedures prefer to operate against normalized data, which tends to be tall and narrow, and often contains classification variables that are used to identify individual rows. Data in non-normal form tends to have one column for each level of one of the classification variables. Converting between the two forms can be accomplished using PROC TRANSPOSE or from within the DATA step. This is one of the 'classic' uses of arrays in conjunction with DO loops.

**NORMAL to NON-NORMAL (Rows to Columns)** (2.4.2)

In the normal form data shown to the right, there are two classification variables SUBJECT and VISIT with one row for each combination. We would like to transpose this data so that there is one row per SUBJECT and one column for each value of VISIT. Notice that subject 208 did not have a VISIT=3 and the last visit for this subject in the study was 10 (out of a possible 16 visits).

Commonly the process of transposing will involve the use of an array and an iterative DO loop.

Normal Form

| Obs | SUBJECT | VISIT | sodium |
|---|---|---|---|
| 1 | 208 | 1 | 13.7 |
| 2 | 208 | 2 | 14.1 |
| 3 | 208 | 4 | 14.1 |
| 4 | 208 | 5 | 14.1 |
| 5 | 208 | 6 | 13.9 |
| 6 | 208 | 7 | 13.9 |
| 7 | 208 | 8 | 14.0 |
| 8 | 208 | 9 | 14.0 |
| 9 | 208 | 10 | 14.0 |
| 10 | 209 | 1 | 14.0 |

*. . . . portions of the table are not shown . . . .*

| Obs | SUBJECT | visit1 | visit2 | visit3 | visit4 | visit5 | visit6 | visit7 | visit8 | visit9 | visit10 | visit11 | visit12 | visit13 | visit14 | visit15 | visit16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 208 | 13.7 | 14.1 | . | 14.1 | 14.1 | 13.9 | 13.9 | 14 | 14 | 14.0 | . | . | . | . | . | . |
| 2 | 209 | 14.0 | 14.0 | . | 13.9 | 14.2 | 14.5 | 13.8 | 14 | . | 13.8 | 14 | 14.1 | 14.2 | 14.1 | 14 | 14.1 |

*. . . . portions of the table are not shown . . . .*

```
data lab_nonnormal(keep=subject visit1-visit16);
   set lab_chemistry(keep=subject visit sodium);
   by subject;
   retain visit1-visit16 ; ❶
   array visits {16} visit1-visit16; ❷
   if first.subject then do i = 1 to 16; ❸
      visits{i} = .;
   end;
   visits{visit} = sodium; ❹
   if last.subject then output lab_nonnormal; ❺
   run;
```

❶ ❷ The new variables (VISIT1 through VISIT16) are defined and retained.

❸ An iterative DO loop is used to clear the array for each subject.

❹ The value to be transposed is assigned to the new variable using VISIT as the index variable to the array.

❺ After all the visits for this subject have been accumulated in the array, the new observation is written.

**NON-NORMAL to NORMAL (Columns to Rows)** (2.4.2)
The process for transposing from columns to rows is very similar to the one shown in the previous example. The primary difference is in the placement of the OUTPUT statement within the DO loop.

```
data lab_normal(keep=subject visit sodium);
   set lab_nonnormal(keep=subject visit:); ❻
   by subject;
   array visits {16} visit1-visit16; ❼
   do visit = 1 to 16; ❽
      sodium = visits{visit}; ❾
      output lab_normal; ❿
   end;
   run;
```

❻ A list abbreviation is used to name all variables that start with VISIT.
❼ The array specifies variables that already exist.
❽ The iterative DO loop steps through the 16 potential visits.
❾ The value is assigned using VISIT as the array index.
❿ The new observation is written from within the DO loop.

**Array Functions** (3.10.3)
There are three functions that have been specifically designed to work with arrays.

- DIM returns the dimension of the array
- LBOUND returns the lowest bound of the array index
- HBOUND returns the upper bound of the array index

The DIM function can be useful when you need to step through an array and you do not know how many elements the array contains.

```
data newchem(drop=i);
   set advrpt.lab_chemistry
          (drop=visit labdt);
   array chem {*} _numeric_; ❶
   do i=1 to dim(chem); ❷
      chem{i} = chem{i}/100;
   end;
   run;
```

❶ The array CHEM contains all numeric variables, but the dimension will depend on how many numeric variables are in the incoming data set. ❷ The DIM function allows us to step through the array without knowing the number of numeric variables.

In this example the index variable (i) ranges from 1 to the dimension. This approach would not work if the index variable did not start at 1.

The HBOUND and LBOUND functions can be used to determine the range of the index variable. In this example we want to find all subjects that are within an inch of any given subject. The first DO UNTIL loop is used to read the height for each individual into the HEIGHTS array. A second pass of the data is made in the second DO UNTIL where the height of each subject is compared to each of the other subjects.

```
data CloseHT;
array heights {&lb:&hb} _temporary_; ❸
do until(done);
   set advrpt.demog(keep=subject ht) end=done;
   heights(subject)=ht; ❹
end;
done=0;
do until(done);
   set advrpt.demog(keep=subject ht) end=done;
   do Hsubj = lbound(heights) to hbound(heights); ❺
      closeHT = heights{hsubj};
      if (ht-1 le closeht le ht+1)
          & (subject ne hsubj) then output closeHT;
   end;
end;
stop;
run;
```

❸ The temporary array is defined using the lowest (&LB) and highest (&HB) subject number of interest.
❹ The subject number itself is used as the index to the array.
❺ The DO loop uses the LBOUND and HBOUND functions to return the limits of the array.

## WORKING ACROSS OBSERVATIONS (3.1)

Because SAS reads one observation at a time into the PDV, it is difficult to 'remember' the values from an earlier observation (look-back) or to anticipate the values of a future observation (look-ahead).
Without doing something extra, only the current observation is available for use.

A form of look-back not discussed in this paper is accomplished through the use of the LAG function. As these techniques typically require neither arrays or DO loops they have not been included in this paper.

### Processing Within Groups (Using FIRST. and LAST.) (3.1.1)

A very common solution to processing within and across groups is the use of FIRST. and LAST. processing. These techniques are especially useful when counting items, but they do require that the data be sorted (to allow the use of the BY statement ❶).

```
data counter(keep=region clincnt patcnt);
   set regions(keep=region clinnum);
   by region clinnum; ❶
   if first.region then do; ❷
      clincnt=0;
      patcnt=0;
   end;

   if first.clinnum then clincnt + 1; ❸
   patcnt+1; ❹

   if last.region then output; ❺
   run;
```

In this example we need to count both the number of clinics and the number of patient visits within a region.
❷ The counters are initialized for each region.
❸ The clinic is counted once for each region, while each patient visit ❹ is counted.
❺ Once all observations for the reagion have been processed the observation can be written out to the data set WORK.COUNTER.

The IF statement at ❸ could be replaced by a simple, and quicker, assignment statement.

```
clincnt + first.clinnum;
```

### Transposing to Temporary Arrays (3.1.2)

When we need to work with values from multiple observations it is often very effective to save the values to an array. Essentially we are transposing the rows to an array and the code is similar to that used in the transpose example earlier in the paper (transposing rows to columns).

In this example we want to calculate the average number of days between visits.

A temporary array ❶ is used to hold all of the visit dates for each subject. The array is cleared ❷ using the CALL MISSING routine. ❸ The values for each observation are loaded into the array and when the last observation for the subject has been read ❹, the values in the array can be processed. The difference ❺ is accumulated ❻ so that the average difference can be calculated ❼.

```
data labvisits(keep=subject count meanlength);
   set advrpt.lab_chemistry;
   by subject;

   array Vdate {16} _temporary_; ❶
   retain totaldays count 0;

   if first.subject then do;
      totaldays=0;
      count = 0;
      call missing(of vdate{*}); ❷
   end;
   vdate{visit} = labdt; ❸
   if last.subject then do; ❹
      do i = 1 to 15;
         between = vdate{i+1}-vdate{i}; ❺
         if between ne . then do;
            totaldays = totaldays+between; ❻
            count = count+1;
         end;
      end;
      meanlength = totaldays/count; ❼
      output labvisits;
   end;
   run;
```

### Building a FIFO Stack (3.1.7)

When processing across a series of observations for the calculation of statistics, such as running averages, a stack can be helpful. A stack is a collection of values that have automatic entrance and exit rules. Values tend to rotate through a stack.

Stacks come in two basic flavors; First-In-First-Out, FIFO, and Last-In-First-Out, LIFO. In a FIFO stack the oldest value in the stack is removed to make room for the newest value. When done correctly implementation of a stack in the DATA step is straightforward.

A three day moving average of potassium levels is to be calculated for each subject. The key will be to designate a temporary array to be used as the stack. The index of this array will start at 0. For a moving average the dimension of the array ❷ and the second argument to the MOD function ❺ will always be the number of items to be included in the moving average.

```
data Average(keep=subject visit labdt
                  potassium Avg3day);
   set labdates;
   by subject;

   * dimension of array is number of
   * items to be averaged;
   retain visitcnt .; ❶
   array stack {0:2}  _temporary_; ❷
   if first.subject then do;
      call missing(of stack{*}); ❸
      visitcnt=0;
   end;
   visitcnt+1; ❹
   index = mod(visitcnt,3); ❺
   stack{index} = potassium; ❻
   avg3day = mean(of  stack{*}); ❼
   run;
```

❶ Items in the temporary array are automatically retained.
❷ The index for the temporary array that is to serve as the FIFO stack always starts at 0.
❸ The stack is cleared for each subject as is the visit counter.
❹ A visit counter is incremented for each visit.
❺ The MOD function is used to determine the index for the array. Because of the cyclic nature of the MOD function, the newest entry will always automatically overwrite the oldest entry❻ in the array (the FIFO stack). This is the key to the processing of the stack.
❼ The mean of the items in the stack are calculated.

## USING SET STATEMENT OPTIONS (3.8)

Although a majority of DATA steps use the SET statement, few programmers take advantage of its full potential. The SET statement has a number of options that can be used to control how the data are to be read. Some of these options include:

- END= used to detect the last observation from the incoming data set(s)
- KEY= specifies a index to be used when reading
- INDSNAME= used to identify the current data source
- NOBS= number of observations
- POINT= designates the next observation to read
- UNIQUE used with KEY= to read from the top of the index

The END= option was used previously to control the exit from a DO UNTIL loop in the example showing the use of the LBOUND and HBOUND functions.

### Using POINT= and NOBS= (3.8.1)

The SET statement by default reads one observation after another, first observation to last. The POINT= option makes it possible to perform a non-sequential read.

The POINT= option identifies a temporary variable that indicates the number of the next observation to read. The NOBS= option also identifies a temporary variable, which after DATA step compilation will hold the number of observations on the incoming data set.

In this example a random subset of a larger data set is to be selected without replacement (each observation can only be selected at most one time). The selected observation will be noted in the temporary array OBSNO by using the observation number as the array index.

```
%macro rand_wo(dsn=,pcnt=0);
   data rand_wo(drop=cnt totl);
      totl = ceil(&pcnt*obscnt); ❶
      array obsno {10000} $1 _temporary_; ❷

      do until(cnt = totl);
         point = ceil(ranuni(0)*obscnt); ❸
         if obsno{point} = ' ' then do; ❹
            set &dsn point=point nobs=obscnt; ❺
            output rand_wo;
            obsno{point}='x'; ❻
            cnt+1;
         end;
      end;
      stop; ❼
      run;
%mend rand_wo;
%rand_wo(dsn=advrpt.demog,pcnt=.3)
```

❶ The percentage of the total number of observations is calculated.
❷ The temporary array must have a dimension that is at least as big as the observation count.
❸ Generate a random number between 1 and the number of observations (OBSCNT).
❹ If the observation number held in POINT has not already been selected , read and write it.
❺ The POINT= and NOBS= options are specified on the SET statement.
❻ This observation is marked as having been selected.
❼ When using the POINT= option the DATA step must be terminated with a STOP statement.

## USING THE DOW LOOP (3.9.1)

Although thought to have been initially proposed by Don Henderson, the DOW loop, which is also known as the DO-W loop, was named for Ian Whitlock who popularized the technique and was one of the first to demonstrate its efficiencies.

### The DATA Step's Implied Loop

The DATA step has an implied loop. This loop is executed once for each incoming observation. During the DATA step's execution phase the DATA statement ❶ is executed at the top of the implied loop. At this time a number of operations are performed on the PDV. These include setting derived variables to missing and incrementing the temporary variable _N_.

```
data implied; ❶
  set big;
  output implied;
  run;
```

### A Single Pass DATA Step

When using a DOW loop, the implicit loop of the DATA step is replaced with an explicit one (here a DO UNTIL loop). Because of the DO UNTIL (DOW) loop, the DATA statement is executed only once, and this can increase the efficiency of the DATA step.

```
data dowloop;
  do until(eof); ❷
     set big end=eof;❸
     output dowloop;
  end;
  stop; ❹
  run;
```

❷ The DO UNTIL loop executes until the temporary variable EOF takes on the value of 1.
❸ The SET statement includes the use of the END= option. The temporary variable EOF will be zero for all observations except the last one.
❹ Although not needed in this example, it is generally a good idea to use a STOP statement to terminate a DATA step that includes a DOW loop.

In the next slightly more interesting example of a DOW loop two data sets are merged using two SET statements. In this case we want to merge a single observation summary data set (which contains a mean value) onto the analysis data, which is then used to calculate a percent change.

```
proc summary data=advrpt.demog;
   var wt;
   output out=means mean=/autoname;
   run;
data Diff1;
   if _n_=1 then set means(keep=wt_mean); ❺
   set advrpt.demog(keep=lname fname wt); ❻
   diff = (wt-wt_mean)/wt_mean;
   run;
```

```
data Diff2;
   set means(keep=wt_mean); ❼
   do until(eof);❽
      set advrpt.demog(keep=lname fname wt)
          end=eof;❾
      diff = (wt-wt_mean)/wt_mean;
      output diff2;
   end;
   stop; ❿
   run;
```

❺ The summary data set is read only once (`IF _N_=1`), however because of the DATA step's implied loop, the IF statement's expression is evaluated for every observation on the analysis data set.
❻ The analysis data set is read and the percent difference from the mean weight is calculated.

❼ Because the implied loop has been circumvented, this SET statement will execute only once.
❽ A DO UNTIL is used to create the DOW loop with the exit criteria being the reading of the final observation.
❾ The END= option is used to create the temporary Boolean variable, EOF, to flag the last observation.
❿ Unless you are very comfortable with your understanding of the operation of the DATA step and what causes it to terminate, the STOP statement is recommended when using a DOW loop.

### Key Indexing – a Simple Hash (6.7.2)

Key Indexing is generally considered the fastest form of table lookups. This technique named and promoted by Paul Dorfman is used to essentially merge two data sets. Unlike the MERGE statement such as the one shown to the right, Key Indexing does not require that either data set be sorted. In this example the clinic name needs to be merged onto the larger data set, WORK.DEMOG. The only variable in common is the clinic number (a numeric value stored in a character variable). Generally sorting these two data sets and applying the MERGE in a DATA step will be fast enough. However when sorting becomes problematic as the data sets get large, or if you just do not want to sort either data set, key indexing may be an option.

```
data clinnames;
   merge demog
         clinicnames;
   by clinnum;
   run;
```

The key to this technique is the use of a temporary array to hold the values that are to be transferred (the clinic names). As a bonus neither data set needs to be sorted.

```
data clinnames(keep=subject lname fname clinnum clinname);
  array chkname {999999} $35 _temporary_; ❶
  do until(allnames); ❷
     set advrpt.clinicnames end=allnames;
     chkname{input(clinnum,6.)}=clinname; ❸
  end;
  do until(alldemog);
     set advrpt.demog(keep=subject lname fname clinnum) ❹
                end=alldemog;
     clinname = chkname{input(clinnum,6.)}; ❺
     output clinnames;
  end;
  stop;
  run;
```

❶ The range or dimension of the array must be sufficient to hold all the clinic names. Since the clinic numbers are used as the array index, and these are probably not sequential, there will likely be a lot of empty space in the array. Because arrays are held in memory and since we generally have quite a bit of available memory, very large arrays are generally not a problem.
❷ A DO UNTIL is used to read the values of the smaller data set so that the clinic names can be placed into the temporary array.
❸ The clinic name is stored in the temporary array using the clinic number as the array index. In this example the clinic number is stored in a character variable so it must be converted to numeric using the INPUT function.
❹ The observation is read that contains the current clinic number. This number is used in an assignment statement ❺ to retrieve the clinic name.

### Hash Object Initialization (6.7.2)

Hash objects can also be used to perform a table lookup. These techniques can be more flexible than key indexing as they do not require a single numeric index. Like key indexing, the hash object also avoids the need to sort the incoming data.

Hash objects need to be declared and defined. Very often this is done in a conditionally executed DO block within the DATA step. This DO block can generally be avoided through the use of a DOW loop.

```
If _n_=1 then do;
   declare hash lookup . . .
   . . . .
end;
```

In the following DATA step the same table lookup, that was performed using key indexing in the previous example, is performed using a hash object. A DOW loop ❻ is used to read the data (the larger data set) and to retrieve a clinic name from the hash object.

```
data hashnames(keep=subject clinnum clinname lname fname);
    if 0 then set advrpt.clinicnames; ❶
    declare hash lookup(dataset: 'advrpt.clinicnames', ❷
                        hashexp: 8); ❸
   lookup.defineKey('clinnum'); ❹
   lookup.defineData('clinname'); ❺
   lookup.defineDone();

   * Read the primary data;
   do until(done); ❻
      set advrpt.demog(keep=subject clinnum lname fname) ❼
           end=done; ❽
      if lookup.find() = 0 then output hashnames; ❾
   end;
   stop;
   run;
```

❶ The attributes of the variables that are to be loaded into the hash object are added to the PDV. This makes it unnecessary to specify the attributes explicitly. This SET statement is never executed and is therefore only used during step compilation to load variable attributes onto the PDV.
❷ The data set containing the clinic names is loaded into the hash object using the DATASET: constructor.
❸ The hash objects size is specified using the HASHEXP: constructor.
❹ The clinic number is defined as the key variable for this hash object (you can have multiple key variables).
❺ The clinic name variable (CLINNAME) is stored in the LOOKUP hash object as a variable that can be retrieved.
❻ The DOW loop is set up using a DO UNTIL. The exit criteria is based on the detection of the last observation on the incoming data set by using the END= option on the SET statement.
❼ The variables of interest, including the index variable CLINNUM, are read from the incoming data set.
❽ Continue to read observations until the last observation (DONE=1), this terminates the DOW loop ❻.
❾ If the FIND method is successful (the clinic number is in the hash object), the clinic name is retrieved from the hash object, and the observation is written to WORK.HASHNAMES.

## SUMMARY

There is so very much that one can do with both DO loops and ARRAYS. This becomes even truer when they are used in conjunction with each other. You need to study and learn the various forms of the statements and you must learn how they are compiled and executed within the context of the DATA step.

You can do a great deal with the simple forms of these statements, but as you learn the deeper nuances of the techniques that surround the use of DO loops and arrays, a new world of opportunity is made available to you.

The tools and techniques associated with the use of DO loops and arrays are a staple of the innovative SAS programmer.

## ABOUT THE AUTHOR

Art Carpenter's publications list includes five books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other user group conferences. Art has been using SAS® since 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.


## AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

## REFERENCES

Many of the examples in this paper have been borrowed (with the author's permission) from the book Carpenter's Guide to Innovative SAS® Techniques by Art Carpenter (SAS Press, 2012).

## TRADEMARK INFORMATION