Paper 354-2009

If I Only Had a Map: Getting From Here to There
Common Approaches to Solving DATA Step Logic Issues

Kim Wilson, SAS Institute Inc., Cary, NC

ABSTRACT

How often have you needed to reshape your data so that multiple observations for the same BY group become condensed into a single observation? Have you wondered how to best approach putting two data sets together in order to perform calculations or to compare values between them? Perhaps you're asked to determine if values or dates overlap between observations, which could have a great impact on your company's production or spending.

Even with a thorough knowledge of SAS® syntax, knowing how to approach such logic issues is the key to efficient programming. Knowing what you have and what you desire to obtain from the data is the first step in tackling this issue. This paper offers tips and techniques to enable you to take a project from start to finish as efficiently as possible.

INTRODUCTION

This paper is written for the intermediate DATA step user who is interested in learning how to approach the logic for a computation from one or more data sets. There are alternative approaches using various SAS procedures to accomplish the same or a portion of a task. However, this paper addresses logic from a DATA step perspective. There are many ways to approach a single task in SAS. The methods that are shown in this paper do not imply that other methods could not be used to achieve the desired outcome. The example code is executed with SAS 9.1.3 or newer software releases, and features that are unique to SAS 9.2 are noted.

Various situations can arise that require a SAS user to ponder which path to take to obtain the desired result. A SAS function, arithmetic operation, or assignment statement can provide a quick solution. Most often, however, a task requires a sophisticated combination of these building blocks. The possibilities for manipulating data and the route to take for each possibility are endless. This paper covers some of the most common situations SAS customers present to SAS Technical Support:

- obtaining counts when certain conditions occur
- assigning non-missing values to missing values within the same observation
- collapsing multiple observations with the same BY value into a single observation
- looking before and after a current observation
- changing variable values to variable names
- creating summary data for multiple observations
- using table lookup techniques with multiple data sets
- exploring how the MERGE statement works when there are and aren't duplicate BY values
- using a hash table

Let's look at some common approaches to working with your data to obtain the desired output.

WORKING WITH A SINGLE DATA SET

When you need summary information from a single data set, some pertinent questions to ask yourself are:

- Is the data set sorted? If not, does it need to be? If so, is it too large to sort?
- Will I consider each observation independently from all others?
- Will I perform the same task on more than one variable?
- Do I need totals or other calculations within a BY group?
- Will I compare observations within a BY group to other observations in the same group?
- Will I need to make a decision about a given observation only after comparing it with others?
- Will any of my values need to become part of new variable names?

Let’s look at a code sample that creates a data set containing doctor visits for six time periods for diabetic patients. Regular visits are needed to check glucose levels in case the current medication needs to be adjusted. If a patient misses a scheduled monthly appointment, a missing value is recorded.

```
data one;
   input pt_id level1 level2 level3 level4 level5 level6 ;
datalines;
254 85 90 . . 101 82
577 88 80 99 . 100 101
231 85 99 100 . 101 99
333 90 . . 101 101 102
580 80 82 . . . .
221 100 . . . . .
;
run;
```

Suppose you want to know how many times patients missed visits 2 and 3 (level 2 and level 3). A variable is needed to keep a cumulative count of when this condition is true. Each observation contributes to the overall total, but each observation is evaluated independently of the other observations. If the condition is met, increment the counter. The final count is on the last observation.

Because you’re interested in just two variables, an IF statement tests the condition of both variables having missing values and, when true, adds 1 to the existing value of COUNT and automatically retains that value from one iteration to another. The syntax "COUNT+1" is called a SUM statement. The variable COUNT is initialized to 0 at the beginning of the step and retained by default, as shown in the following sample code and output:

```
data two;
   set one;
   if level2=. and level3=. then count+1;
run;
```

```
Obs    pt_id  level1  level2    level3    level4    level5    level6    count
 1      254     1       2         .         .         7         8         0
 2      577     1       2         5         .         7         8         0
 3      231     1       2         5         .         8         9         0
 4      333     1       .         .         7         8         9         1
 5      580     1       2         .         .         .         .         1
 6      221     1       .         .         .         .         .         2
```

Had an ASSIGNMENT statement been used instead of a SUM statement, a RETAIN statement would have also been needed to prevent the variable from being reset to missing with each iteration of the DATA step. Assigning an initial value for COUNT is also necessary to prevent a missing value from being introduced on the first iteration of the step. This initial value is made in the RETAIN statement by placing the 0 value after the variable COUNT as follows:

```
   retain count 0;
   if level2-. And level3=. then count=count+1;
```

USING DO LOOPS AND ARRAYS

Let’s take the example one step further. You need to assign the last non-missing glucose value to any variables on that observation with missing values. Because each observation is considered independently of all others, no sorting is required. In order to process each variable the same way without hard coding the names, an iterative DO loop is used to move through each element of an array. An array is a group of variables that can be processed one element at a time for the duration of the current DATA step.

The colon modifier used in the ARRAY statement is a shortcut method for referencing each variable with the given prefix. In this case, each variable that begins with "level" is selected for the array.

The CONTINUE statement stops processing the current DO-loop iteration and resumes processing the next iteration. This is helpful when you don’t need the current observation to continue to be evaluated by other code further down in the same DO loop. In this code, CONTINUE isn’t necessarily beneficial because it’s inside an IF-THEN statement and the only remaining code is an ELSE statement, which would not execute anyway. It was included in the following example for illustration purposes.

```
data two(drop=i hold );
   set one;

   array t(6) level: ;

   do i=1 to 6 ;
   if t(i) ne . then do;
      hold=t(i);
      continue;
   end;
   else t(i)=hold;
end;
run;
```

```
Obs    pt_id    level1    level2    level3    level4    level5    level6
 1      254       85        90        90        90       101        82
 2      577       88        80        99        99       100       101
 3      231       85        99       100       100       101        99
 4      333       90        90        90       101       101       102
 5      580       80        82        82        82        82        82
 6      221      100       100       100       100       100       100
```

The previous example was straightforward because none of the observations needed information from another observation. The array saved much time and coding because all variables were referenced by the array name and the index variable that called them by their numbered order. Customers often use a macro to accomplish the preceding task rather than using the array and iterative DO loop. This can result in a large amount of DATA step source code that is easily avoided by using arrays and DO loops.

WORKING WITH MULTIPLE OBSERVATIONS FOR THE SAME BY GROUP

The following example is one of the most frequently asked questions from customers requesting help from SAS Technical Support. Consider a situation in which multiple observations exist for the same BY variable. For ease of reading, your manager asks you to collapse multiple observations for student grades into a single observation per student, as shown in the following sample code:

```
data students;
   input name $ class $ grade ;
datalines;
anne math 98
anne science 88
anne english 87
anne music 90
mike math 100
mike science 99
mike english 97
;
run;
```

Because it's necessary to process each BY group as a unit, BY-group processing should come to mind. Place the variable that defines the logical grouping of observations in the BY statement. The BY statement is required to allow the use of the automatic variables FIRST.NAME and LAST.NAME. These variables are set to 1 when the first or last observation, respectively, in a BY group is read. These variables make it possible to execute code when a new BY group begins and when it ends. All values within the same BY group are loaded into arrays as each observation in that group is processed. At the end of the BY group, the observation is written to the data set.

Because you want to collapse the observations that have the same BY values, in this example NAME, you need to sort the data by NAME. The example data is already sorted; therefore, the SORT procedure isn't needed. There are two variables in addition to the BY variable, therefore, each of them needs a variable for each value in a given BY group. A BY group is the logical grouping of the same values for the variables used in the BY statement. An array is the perfect tool because it's a temporary grouping of variables and it's simple to refer to each variable by its location in the array. An array exists only for the duration of the given DATA step, which is different from some other computer languages.

You need to know the number of observations in the largest BY group so that all arrays are given the same number of elements and that this number isn't too high or low based on a quick scan through the data. A direct way is to run the FREQ procedure. The ORDER= option causes the largest BY group to be shown first. The NOCUM option prevents cumulative values from being output as shown in the following code sample and PROC FREQ output:

```
proc freq data=students order=freq;
   tables name/ nocum;
run;
```

```
name      frequency     percent
anne          4          57.14
mike          3          42.86
```

```
data out (keep=name subj1-subj4 score1-score4);
   retain subj1-subj4 score1-score4;
   array subj (4)$ subj1-subj4;
   array score (4)  score1-score4;
   set students;

   by name;

   if first.name then do;
      i=1;
      call missing(of subj(*),of score(*));
    end;
   subj(i)=class;
   score(i)=grade;

   if last.name then output;
   i+1;
run;
```

You can see from the PROC FREQ output that four observations per BY group is the maximum; therefore, each array has four elements. The names must be either variables that you define in the ARRAY statement or variables that SAS creates by concatenating the array name and a number of that element. In this example, had the array names not been explicitly written, SAS would have assigned them the same names. They were included in this example for illustration purposes.

Without a RETAIN statement, SAS automatically sets variables that are assigned values by the ASSIGNMENT statement to missing before each iteration of the DATA step. If the RETAIN statement was omitted, each new iteration of the DATA step causes all variables to be reset to missing. Although the ASSIGNMENT statement assigns values from each observation to the appropriate arrays, the new observation is written to the data set only when the last observation for each BYgroup is read.

Some BY groups have fewer than four observations, so it’s necessary to set all array variables to missing so that variables from BY groups with more observations than the previous BY group don’t carry values over to the current BY group. A variable is needed as a counter so that the appropriate array elements are filled. The variable I is set to 1 at the beginning of each new BY group so that the first observation in the BY group populates the first element of each array.

Each array is populated by the appropriate variable using ASSIGNMENT statements. When the last observation per BY group has populated each array, the observation is written to the data set.

The counter variable I is incremented by 1 per the SUM statement. This automatically retains the value from one iteration to the next, as shown in the following output:

```
Obs    subj1    subj2     subj3     subj4    score1    score2    score3    score4    name
 1     math     science   english   music      98        88        87        90      anne
 2     math     science   english             100        99        97         .      mike
```

LOOKING BEFORE AND AFTER A CURRENT OBSERVATION

Situations can occur in which you might need to make a decision about a current observation based on the values from the previous observation or the observation that follows. The LAG function enables access to the previous observation but there isn’t a function to enable you to look ahead. The following example shows how to accomplish this by reading the next observation with the POINT= option on the SET statement. It’s common to use the POINT=

option to accomplish a table lookup so values from one data set are used with another data set. This example uses one data set rather than two.

```
data sales;
   input company $ month_end :mmddyy8. sales;
   format month_end mmddyy10.;
datalines;
abc 01312008 10000
abc 02282008 25000
abc 05312008 15000
abc 10312008 12000
abc 12312008 20500
xyz 02282008 30000
xyz 03312008 10000
xyz 07312008 12500
xyz 10312008 15000
xyz 11302008 20000
xyz 12312008 11000
;
run;
```

The LAG function obtains the value for the previous observation. In order to avoid unexpected results, never create a variable with a LAG function in a conditional statement, such as an IF-THEN/DO block. Many SAS programmers assume that a LAG function is going to add a value to the lag queue every time the implicit loop iterates. If the LAG function doesn't execute, it doesn't add a value to the queue, thus the confusion when the LAG function is used in an IF or DO loop. Remember that a BY statement gives access to FIRST. and LAST. variables so you can use these when code should execute only on specific observations. In this example, you do not want to obtain the previous observation's value when reading the first observation in a BY group. That would produce the last observation in the previous BY group. Additionally, you wouldn't want to obtain the next observation when you're reading the last observation in a BY group. FIRST. and LAST. variables make this process easier to code.

When reading data from multiple observations, like-named variable values overwrite those that are already in the Program Data Vector (PDV). This is why it's helpful to use a KEEP= option (or DROP= option, whichever produces the shortest list of variables) on the second SET statement so that only necessary variables come into the PDV, as shown below. It's also crucial to use the RENAME= option to rename common variables to new names that don't exist in the PDV.

```
data totals;
   set sales;
   by company;
   lg=lag(sales);
   if not first.company then last_mo=sales-lg;
   if not last.company then do;
      next=_n_+1;
      set sales(keep=sales rename=(sales=sales2)) point=next;
      next_mo=sales2-sales;
   end;
run;
```

```
Obs    company    month_end    sales    sales    last_mo    sales2    next_mo

  1      abc     01/31/2008    10000        .          .     25000      15000
  2      abc     02/28/2008    25000    10000      15000     15000     -10000
  3      abc     05/31/2008    15000    25000     -10000     12000      -3000
  4      abc     10/31/2008    12000    15000      -3000     20500       8500
  5      abc     12/31/2008    20500    12000       8500     20500          .
  6      xyz     02/28/2008    30000    20500          .     10000     -20000
  7      xyz     03/31/2008    10000    30000     -20000     12500       2500
  8      xyz     07/31/2008    12500    10000       2500     15000       2500
  9      xyz     10/31/2008    15000    12500       2500     20000       5000
 10      xyz     11/30/2008    20000    15000       5000     11000      -9000
 11      xyz     12/31/2008    11000    20000      -9000     11000          .
```

In the preceding data set, all months from January through December are not represented for each company. If you need to collapse this data set so that each company's information is on one observation, it's also necessary to have each month's sales stored in the appropriate variable name. This differs from the example in the section, "Working with Multiple Observations for the Same By Group," where the grades were put into new variables without the variable names having significant positions.

An array is still the method of choice for storing the new variables; however, because these values are dates, the month number is extracted with the MONTH function to determine which array element receives the appropriate sales figure for that month. Because the new observation is written to the TOTALS data set after the last observation per BY group is read, the KEEP= option is used to specify the BY variable and the new variables.

Unless variable names are specified in the ARRAY statement, the names of the variables are the array name followed by the number of the array element (for example, DATES1, DATES2,…DATES12) as shown in the following sample code and output:

```
data totals(keep=company dates1-dates12);
   set sales;
   by company;
   array dates (12);
   retain dates1-dates12;
   mon=month(month_end);
   if first.company then do;
      call missing(of dates(*));
   end;
   dates(mon)=sales;
   if last.company then output;
run;
```

```
     c                                                              d       d       d
     o      d       d       d     d     d     d     d     d   d     a       a       a
     m      a       a       a     a     a     a     a     a   a     t       t       t
     p      t       t       t     t     t     t     t     t   t     e       e       e
O    a      e       e       e     e     e     e     e     e   e     s       s       s
b    n      s       s       s     s     s     s     s     s   s     1       1       1
s    y      1       2       3     4     5     6     7     8   9     0       1       2

1   abc   10000   25000       .   .   15000   .       .   .   .   12000       .   20500
2   xyz       .   30000   10000   .       .   .   12500   .   .   15000   20000   11000
```

This output puts each month's sales into the appropriately numbered variable by month, but it's not very descriptive. It would be more meaningful if the month names were displayed as column headings rather than D1-D12. In the next example, a macro is used to generate labels that contain the month names for the variables. Macro code is needed so that the variable names and their associated labels are generated dynamically with a macro %DO loop. If this approach wasn't used, the LABEL statement would have to be hard coded twelve times. Macro is a text substitution language, thus it's perfect for this situation. The LABEL option is required for the PRINT procedure in order to see the labels.

For programmers who are unfamiliar with the macro language, a macro begins with %MACRO followed by the macro name. SAS code is put inside a macro and, when finished, a %MEND statement marks the end of the macro. To invoke the macro, a % sign followed by the name of the macro sets it in motion.

In regular DATA step code, the PUT function assigns a format to a variable, but the PUTN function is used instead for macro variables. Also, %SYSFUNC is required to preface the function. The combination of this syntax produces the DATA step variable name, an "=", and the new label value as shown below. Because there isn't a SAS-supplied format to produce the month name from digits 1-12, the FORMAT procedure is needed to create the format.

```
proc format;
   value mon 1=”Jan”
             2=”Feb”
             3=”Mar”
             ...
            12=”Dec”;
run;

%macro test;
data totals(keep=company d1-d12);
   set sales;
   by company;
   array d (12);
   retain d1-d12;
   mon=month(month_end);
   label
      %do i=1 %to 12;
          d&i="%sysfunc(putn(&i,mon.))"
      %end;;
   if first.company then do;

     call missing(of d(*));
   end;
   d(mon)=sales;
   if last.company then output;
proc print label;
run;
%mend test;

%test
```

```
Obs   company    Jan     Feb     Mar    Apr    May    Jun    Jul    Aug   Sep    Oct     Nov     Dec

 1      abc     10000   25000     .      .    15000    .       .      .     .    12000      .    20500
 2      xyz         .   30000   10000    .        .    .    12500     .     .    15000   20000   11000
```

CHANGING VARIABLE VALUES TO VARIABLE NAMES

Often you will want to reshape your data for readability or reporting purposes. For example, if a variable has duplicate observations for the same value, it is easier to read the data if those values become variable names and its values are populated by another variable in the data set. When considering putting values into variable names, make sure that if the values begin with numeric values, you add code to put either an underscore or letter at the beginning so that the variable names meet the SAS name requirements.

The following sample code creates a data set with various product values and the number of contracts for a particular product. Duplicate product values exist. Each unique product value becomes a variable name and its value is the contract value that is found on the original observation.

```
data one;
   input product $ contract;
datalines;
BE2 360
BE2 361
BE3 362
BE4 363
CE1 440
CE1 441
CE1 442
CE2 551
CE2 552
;
run;
```

Each unique value of PRODUCT is put into one macro variable, &VARLIST, and separated by spaces. The total number of those values is put into macro variable &CT. Macro variables can be created inside a DATA step with CALL SYMPUTX, but the SQL procedure is simplest when putting them all into one macro variable. Both of these are needed in the next DATA step when values become variable names, as shown below:

```
proc sql noprint;
   select  distinct(product) into :varlist separated by ' '
   from one;
   select count(distinct product) into :ct
   from  one;

quit;
```

An array is used to hold the value of &VARLIST, which is a list of all unique values of PRODUCT. Because an array is a temporary grouping of variables, these values are assumed to be variable names. Within a DO loop with the macro variable &CT as the upper boundary, this loop runs from 1 to 5. The value of PRODUCT is compared to the variable name of each array element one at a time because of the VNAME function. Without the VNAME function, PRODUCT would be compared to the *values* of each variable in the array rather than the *variable name*. When the condition is true, the array element is set to the value of CONTRACT.

Because CONTRACT is a numeric variable and PRODUCT is a character variable, the PUT function converts CONTRACT to a character variable. By default, the PUT function doesn't left-justify the values after the conversion is made to character so the LEFT function is needed. Otherwise, the value would remain right-justified.

The LEAVE statement stops processing the current loop and resumes with the next statement in the sequence. This is useful because when the current observation's product value finds a match in the array, there's no need to search further. The value of PRODUCT will be found in the array only once, so to save time, exit the loop which ends that iteration of the DATA step and execution returns back to the top of the step as shown below:

```
data output(drop=product contract i);
   set one;
   by product;
   array myvar(&ct) $ &varlist;
   do i=1 to &ct;
      if product = vname(myvar(i)) then do;
         myvar(i)=left(put(contract,8.));
         leave;
      end;
   end;
run;
proc print;
run;
```

```
Obs    BE2    BE3    BE4    CE1    CE2
 1     360     .      .      .      .
 2     361     .      .      .      .
 3      .     362     .      .      .
 4      .      .     363     .      .
 5      .      .      .     440     .
 6      .      .      .     441     .
 7      .      .      .     442     .
 8      .      .      .      .     551
 9      .      .      .      .     552
```

Notice the variable names are now the former values of the variable PRODUCT.

The examples shown thus far have incorporated numerous techniques. These include using BY-group processing, conditional logic, and assigning values to variables accordingly, and keeping up with values from multiple observations.

This next example incorporates many of the features you've seen in earlier examples while accomplishing a task that involves hospital admission data. A patient's admission and discharge dates are recorded on an observation. It's possible that a new observation records the same or following day for admission as the previous observation's discharge date. You want to know when these situations occur so that you can summarize the data and easily consolidate a patient's admission and discharge dates for stays in the hospital. Consider the following sample code:

```
data hospital;
   input pt admit :mmddyy6. discharge  :mmddyy8.;
datalines;
1 081296 08201996
1 082096 08251996
1 110796 11081996
2 070197 07041997
2 090397 09041997
2 030598 03071998
2 030898 03091998
2 030998 03291998
;
run;

data service_dates(drop=admit discharge flag);
   set hospital;
   by pt;
   retain first last flag 0;
   if first.pt then do;
      flag=0;
      first=admit;
      last=discharge;
      return;
   end;

   if flag=0 and (admit=last or admit-1=last) then do;
      last=discharge;
      flag=1;
   end;
   else if flag=0 and (admit ne discharge or admit-1 ne last) then do;
      output;
      first=admit;
      last=discharge;
   end;
   else if flag=1 and (admit=last or discharge-1=last) then do;
      last=discharge;
      output;
   end;
   else if flag=1 and ((admit ne last) or ((admit-1) ne last)) then do;
      output;
      first=admit;
      last=discharge;
      flag=0;
      if last.pt then output;
   end;
   format last first admit discharge mmddyy10.;
run;
```

Because there are multiple observations for the same BY-group value, BY-group processing is used so that FIRST. and LAST. variables aid with conditional processing and getting summary information for each patient. When a new patient group begins, set the ADMIT and DISCHARGE values to new variables and list the new variables on a RETAIN statement so that their values are not reset to missing at the top of the DATA step's new iteration. IF-THEN/ELSE statements check the conditions of various variables to establish when the current observation is a totally new admission or is within the timeframe of a prior admission. Once all observations for an admission have been grouped, the FIRST and LAST variables are set appropriately, as shown in the following output:

```
Pt         First        Last

 1       08/12/1996   08/25/1996
 1       11/07/1996   11/08/1996
 2       07/01/1997   07/04/1997
 2       09/03/1997   09/04/1997
 2       03/05/1998   03/29/1998
```

WORKING WITH MULTIPLE DATA SETS

There are numerous ways to obtain values from more than one data set in the same step. When you need to simply concatenate two or more data sets together, a SET statement accomplishes this task perfectly, as shown in the following code.

```
data final;
   set a b;
run;
```

If the values from each data set should be interleaved, a BY statement after the SET statement returns the output in BY variable order. Because there are multiple data sets, it's often helpful to see the name of the data set on each observation. As of SAS 9.2, the INDSNAME= option provides a straight-forward way to obtain this information as shown in the next step. The option is set to a variable name of your choice and, by default, the variable is of type character with a length of 41. Because this variable isn't written to the output data set, a new variable name of choice, in this case FNAME, should be created in an ASSIGNMENT statement.

```
data final;
   set a b indsname=dsname;
   fname=dsname;
run;
```

When you desire to work with more than one data set, the MERGE, UPDATE, and MODIFY statements are also used for this purpose and all have their unique advantages and disadvantages. The hash object is one of the most versatile new features of Base SAS®, and with SAS 9.2, several additional methods are available to enable you to combine data in ways that weren't possible before with the hash object. The FORMAT procedure is another way to obtain a value from one data set for use in another. The SET statement has options such as KEY= and POINT= that accomplish this task as well.

You're probably asking, "How do I know which to use?" That's a good question, and one we hear often from our customers when they call Technical Support. There are several questions to ask yourself in order to make a good choice:

1. Is the data sorted? If not, do you plan to sort it?

   If the data set is too large to sort based on your computer resources, you might want to consider building an index on the data if you plan to use MERGE, UPDATE, or MODIFY with the KEY= option. An index requires resources to build and maintain, and should be considered only if you'll be using the index for more than the current project and if you'll retrieve 25% or less of the observations. If using a BY statement with a MERGE or an UPDATE statement, the data sets must be sorted; however, using a MODIFY statement with a BY statement doesn't require a sort because a dynamic WHERE statement is generated behind the scenes to match values.

   If the hash object is used, sorting isn't a requirement.

2. How many variables are to be updated with data from another source?

   If you want to assign a value to a new variable based on an existing variable, PROC FORMAT is a good solution as long as the number of ranges doesn't exceed 500,000. If the number of ranges exceeds 500,000, a hash table should be considered to prevent memory issues. Most SAS customers have probably written a custom format using PROC FORMAT and know that it requires a START value and a LABEL value (for example, 1=male 2=female), as shown in the following example:

   ```
   proc format;
      value myfmt 1='male'
                  2='female';
   run;
   ```

   When the format is applied, all values of 1 become 'male' and values of 2 become 'female'. A SAS data set can be used to build a format with the CNTLIN= option with PROC FORMAT, and then later the format is applied to a data set with a FORMAT statement.

   Imagine you have a large data set called A that contains 100 variables that are related to employee information. Here is partial output of that data which is used in the next DATA step.

   ```
   emp_num    f_name    salary    position
     149      jim        50000    accountant
     256      mary       75000    attorney
     165      mike       25000    cook
     209      scott     100000    doctor
   ```

You've been given the assignment of adding each employee's salary to an existing data set called COMPANY, matching on the employee number. Each employee number has a specific salary value and there are never duplicates. This is a perfect scenario for creating a format for the employee number and salary.

Because a format requires START and LABEL values, these are created by renaming EMP_NUM to START and the salary variable to LABEL. A format name must be assigned. PROC FORMAT is used to read the newly created data set, FORMATS, and now the format is ready for use. A new variable is created by reading a variable called EMP_NUM and assigning the formatted value as follows:

```
data formats;
   set a (rename=(emp_num=start salary=label));
   fmtname='emp_sal';
run;

proc format cntlin=formats;
run;

data new;
   set company;
   newvar=put(emp_num,emp_sal.);
run;
```

If there is more than one variable to be updated, another technique is needed. From there, we ask ourselves more questions about the data.

3. Are there duplicates of the key variable that link the two data sets?

If the table that you're updating has duplicates of the key variable, the UPATE statement will not update each duplicate observation. The statement updates only the first observation in the BY group. Normally this isn't desirable for most customers.

This example illustrates a fairly common scenario. Using an UPDATE statement accomplishes the task beautifully based on its design. All of the observations in the transaction data set update the first observation in the master, which gives the desired output as shown in the following code and output.

```
data a;
   input co x1 x2 x3;
datalines;
123 1 . .
123 . 2 .
123 . . 3
456 11 . .
456 . 24 .
456 . . 65
;
run;

proc freq data=a;
   tables co/out=single(keep=co);
run;


OUTPUT
Obs     co
 1     123
 2     456


data new;
   update single a;
   by co;
run;


Obs      co     x1     x2     x3

 1      123      1      2      3
 2      456     11     24     65
```

4. Does an index exist on one of the data sets?

   An index is an optional file that you can create for a SAS data file in order to provide direct access to specific observations. If one exists, it can be used in place of sorting data sets before a MERGE, UPDATE, SET, or MODIFY statement. The KEY= option in the SET statement allows direct access to the matching observation, which prevents a read through each observation in the transaction data set until a match is found. This can save time and resources; however, benchmarking is crucial in order to know which works best for your data. (Numerous samples that illustrate using a SET statement with the KEY= option are located in the DATA step Sample section of the SAS Customer Support site (SAS Institute Inc. 2008a).

   If an index doesn't exist, it's not necessarily best to create one for the sake of using it for combining data sets. An index can be a very useful tool, but there are also costs. Additional CPU time is needed to create, update, and maintain the index when the data file is modified. Additional disk space is required to store the index, and I/O can increase when the index is used.

   There are general guidelines when considering an index for a data set. Read more about this in the "SAS Files" chapter of the *SAS 9.2 Language Reference: Concepts* (SAS Institute Inc. 2008b).

USING A MERGE STATEMENT

The MERGE statement is used widely by SAS customers. It's easy to code, and as long as you know your data and the results that are produced by default, it will probably become the tool that you pull from the box most often.

Earlier in the section, "Looking Before and After A Current Observation," I showed one approach for obtaining the next observation. Many SAS programmers have asked whether a "LEAD function" is available for this situation. Unfortunately, at this time the function is not available. However, a MERGE statement is useful to solve this problem. The same data set, SALES, as shown in the section "Looking Before and After A Current Observation", is used to illustrate this technique. Consider this data set:

```
data lead;
   merge sales sales(firstobs=2 drop=month_end rename=(company=co sales=sales2));
   if company=co then do;
      difference=sales2-sales;
   end;
run;
```

The data set is essentially merged with itself, but the second reference uses the FIRSTOBS= option so that the data set reads observation 2 initially. This causes observation 1 to merge with observation 2, then observation 2 to merge with observation 3, and so on. Notice that the DROP= (or KEEP=) option can be used to limit those variables populating the PDV. Also, it's crucial that all variables that you keep for the PDV get renamed. Otherwise, commonly named variables overwrite those that are in the PDV that came from the first reference to the data set in the MERGE statement. It's also important that you compare the BY-group variable to its renamed counterpart to ensure that you stay within the same BY group. Otherwise, you'll compare the last observation from one BY group to the first observation in the next BY group. You'll commonly hear this technique referred to as "merging a data set with itself".

If there are duplicate values of the BY variable, pay close attention to whether one or both data sets contain duplicates in addition to what type of results you desire. Many SAS programmers assume that values from the second data set listed in the MERGE statement are the values that are always in the final output. That is not the case! It depends on whether there are observations with duplicate values in one or both data sets. The easiest way to anticipate what goes to the newly created data set is to know how the values are loaded into the PDV as the MERGE executes. The PDV is a logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads data values from the input buffer or creates them by executing SAS language statements. The data values are assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set as a single observation.

The data set values are loaded into the PDV based on the order that the data sets are listed in the MERGE statement. Using these data sets, the PDV for each iteration is illustrated as follow:

```
data a;
   input co x1 x2 x3;
datalines;
123 1 10 20
123 30 2 40
333 10 20 30
456 11 12 13
;
run;

data b;
   input co x1 z;
datalines;
123 99 100
456 88 200
456 77 300
;
run;
data merged;
   merge a b;
   by co;
run;
```

```
Obs     co    x1    x2    x3     z

1      123    99    10    20   100
2      123    30     2    40   100
3      333    10    20    30     .
4      456    88    12    13   200
5      456    77    12    13   300
```

The first observation is read from data set A and because variable Z isn't in that data set; it has a missing value.

CO	X1	X2	X3	Z
123	1	10	20	.

The first observation is read from data set B and values from variables that are common to both data sets overlay the values currently in the PDV. The PDV looks like this when the observation has been read:

CO	X1	X2	X3	Z
123	99	10	20	100

This observation is written to the data set MERGED. Control goes back to data set A and another observation is read.

CO	X1	X2	X3	Z
123	30	2	40	100

Notice that the unique variables for a given data set are retained until the end of the current BY group. That's why Z remains in the PDV throughout the BY group.

If another observation for the same BY group had existed, it would be read next. Because there are no more observations with the same BY value, by default all variables in the PDV are reset to missing values.

This observation is written to the data set MERGED. There are no more observations from either data set where CO=123, so all variables in the PDV are set to missing.

The next observation is read from data set A.

CO	X1	X2	X3	Z
333	10	20	30	.

There are no observations that match CO=333 in data set B, so the observation is output to data set MERGED, and all variables in the PDV are set to missing.

The next observation is read from data set A.

CO	X1	X2	X3	Z
456	11	12	13	.

The first observation where CO=456 is read from data set B, and this is how the PDV looks afterwards:

CO	X1	X2	X3	Z
456	88	12	13	200

This observation is output to the data set MERGED.

There aren't any additional observations for the same BY group in data set A so the next observation is read from B.

CO	X1	X2	X3	Z
456	77	12	13	300

Notice that X2 and X3 values remained in the PDV because they are unique to data set A. This observation is written to data set MERGED.

Notice that even though the observation from data set B did not contain values for X2 and X3, those values remain in the PDV because they are unique to data set A and the BY group has not been exhausted. This is a real "gotcha" for many customers because they do not expect this to happen. This is why it is very important to know your data and know how you want the output to look if the BY groups have duplicate observations in one or both data sets.

Essentially, when all observations in the BY group have been read from one data set and there are still more observations in another data set, SAS performs a one-to-many merge until all observations have been read for the BY group. This means the common variables from the data set having the most observations for each specific BY group will be in the output data set.

If you always want to see the values from the second data set for common variables, a simple solution is to drop the common variables from the first data set listed in the MERGE statement. This guarantees that the values come from the second data set. That looks like this:

```
data merged;
   merge a (drop=x1) b;
   by co;
run;
```

If BY groups exist in A that aren't in B, X1 will have missing values in the newly created data set MERGED. This is not acceptable to most customers.

The solution is to rename the common variables in the first data set and use the IN= data set option on both data sets. Then test when an observation is found in the first, and not second, data set so you can set the original variables back to their values

Here's an example using the same data sets A and B above. The variable X1 is renamed to NEWX1 as the data set is read. If a CO value is in data set A and not in data set B, assign NEWX1 back to X1 as follows:

```
data merged;
   merge a(rename=(x1=newx1) in=ina) b(in=inb);
   by co;
   if ina and not inb then x1=newx1;
   drop newx1;
run;
```

```
Obs      co     x2     x3     x1      z

 1      123     10     20     99     100
 2      123      2     40     99     100
 3      333     20     30     10       .
 4      456     12     13     88     200
 5      456     12     13     77     300
```

USING MULTIPLE APPROACHES TO COMBINE DATA

As stated earlier, there are numerous ways to approach any logic-related task. In order to determine which technique is recommended, know your data. If the data set isn't sorted and there are multiple key variables, a hash table should be considered. As you've seen, the MERGE statement is very powerful and one of the most widely used statements in SAS.

In order to illustrate multiple approaches to combining data, two data sets are joined by NAME so that when there are missing FRUIT values from data set A, the value from F in data set B populates the field.

Both data sets do not have a common variable, but it's obvious they should be joined on the field that contains names. Because a MERGE statement is going to be used, both data sets must be sorted on their respective name fields as follows:

```
data a;
   input name $ fruit $;
datalines;
kim apple
jim .
mark .
phil orange
;
run;

data b;
   input name2 $ f $;
datalines;
jake grape
jim tangelo
libby kiwi
jake mango
mark banana
phil lemon
mark lime
;
run;

proc sort data=a;
   by name;
run;

proc sort data=b;
   by name2;
run;
```

In order to merge with a BY statement, one of the data sets must have the variable containing the name values renamed. It doesn't matter which data set is selected for the RENAME= data set option in the MERGE statement. Because the fruit values from data set B should overwrite those from data set A, the fruit field is also renamed.

In this particular instance, we only want those observations from data set A to be in the output data set. The IN= data set option creates a Boolean variable that indicates if the data set contributes to the current observation. The variable name used with the IN= option is a programmer choice, but make sure it's not already used in either data set. In this example, variable A is set to 0 at the beginning of the step and gets a value of 1 when the observation contributes to the merge for that iteration of the step as follows:

```
data new;
   merge a (in=a)b(rename=(name2=name f=fruit));
   by name;
   if a;
run;
```

```
Obs    name    fruit
 1     jim     tangelo
 2     kim     apple
 3     mark    banana
 4     mark    lime
 5     phil    lemon
```

This following DATA step completes a table lookup by reading an observation from data set A with a SET statement followed by a DO loop that reads through data set B one observation at a time.

```
data new(keep=name fruit);
   set a ;
   found=0;
   do i=1 to num;
      set b nobs=num point=i;
      if name=name2 then do;
         found=1;
         fruit=f;
         output;
      end;
      else if i=num and found=0 and name ne name2 then output;
   end;
run;
```

The NOBS= option in the SET statement creates a variable containing the number of observations in that data set. This variable is used as the upper boundary of the DO loop and the POINT= option points directly to each observation by its number. The IF condition checks for matching NAME and NAME2 values and then sets FOUND to a value of 1. This variable is needed so that if a match is never found in data set B, FOUND will still have a value of 0 that can be used on the ELSE condition, so the observation is output as follows:

```
Obs    name    fruit

 1     jim     tangelo
 2     kim     apple
 3     mark    banana
 4     mark    lime
 5     phil    lemon
```

The last approach used with data sets A and B is the hash object. This is a fairly new addition to the DATA step language and some of the new features that are available in SAS 9.2 are used in the following example.

Data set B contains the updated fruit values that belong in the output data set when there is a match on name values. This data set is loaded into a hash table and isn't required to be sorted before loading. The new duplicate key, or MULTIDATA feature, that was introduced with SAS 9.2 is a very significant enhancement to the hash object. This enables duplicate values of the key variable to be loaded and accessed. The key and data variables are defined with the defineKey and defineData methods. A CALL MISSING function establishes initial values for variables because they aren't in the PDV and known to the DATA step yet. Consider the following code:

```
data new;
   length f $8 name2 $8;
   if _n_=1 then do;
      declare hash h(dataset:"b",multidata: 'y');
      h.defineKey("name2");
      h.defineData("f");
      h.defineDone();
      call missing(f,name2);
   end;
   set a(rename=(name=name2));
   rc=h.find();
   if rc=0  then fruit=f;
   output;
   h.has_next(result: r);
   do while(r ne 0);
      r=h.find_next();
      if r=0 then do;
         fruit=f;
         output;
      end;
      h.has_next(result: r);
   end;
run;
proc print data=new;
run;
```

The example uses a DO WHILE loop and the HAS_NEXT and FIND_NEXT methods to loop over duplicate values of NAME2 and return the appropriate F values when a match is found. A match is found with the FIND method and when that occurs, RC is set to a value of 0. The HAS_NEXT method looks for another value of the same key variable (NAME2) and sets R to a non-zero value when that condition is met.

If the hash object is new to you, don't be overwhelmed. There are some great resources in the "Recommended Reading" section that will aid your learning of this new feature. The hash object is a good choice if your lookup data set will fit into memory, the key contains multiple variables of mixed types, and you want to access all information in a BY group at the same time.

CONCLUSION

The DATA step is a powerful mechanism for accomplishing many tasks. The options for accomplishing tasks are endless, and knowing the tools that you have in your tool box and when it's best to use each one helps greatly. Hopefully you've learned some new approaches to solving logic problems that will aid you in accomplishing the various tasks in your programming duties.

RECOMMENDED READING

Ray, Robert, and Jason Secosky. 2008. "Better Hashing in SAS® 9.2". Cary, NC: SAS Institute Inc. Available at support.sas.com/rnd/base/datastep/dot/better-hashing-sas92.pdf.

SAS Institute Inc. 2009. *SAS® 9.2 Base SAS, Base SAS Language Reference Dictionary*. Available at support.sas.com/documentation/cdl/en/lrdict/59540/HTML/default/lrdictwhatsnew902.htm#.

SAS Institute Inc. 2008a. DATA Step Samples. Samples and SAS Notes Web Page. Available at support.sas.com/kb/?ct=51006&qm=3&la=en.

SAS Institute Inc 2008b. *SAS® 9.2. Base SAS, Base SAS Language Reference Concepts*. Available at support.sas.com/documentation/cdl/en/lrcon/59522/HTML/default/a002299817.htm.

Secosky, Jason, and Janice Bloom. 2007. "Getting Started with the DATA Step Hash Object". Cary, NC: SAS Institute Inc. Available at support.sas.com/rnd/base/datastep/dot/hash-getting-started.pdf.

ACKNOWLEDGMENTS

The author thanks Scott McElroy, Russ Tyndall, Sally Walczak, Kevin Hobbs, and Jason Secosky for reviewing this paper.

CONTACT INFORMATION

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

Kim Wilson
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: 919-677-8008
Fax: 919-531-9449
Web: support.sas.com/ts
E-mail: support@sas.com