PAPER IN09_05

The Program Data Vector As an Aid to DATA step Reasoning

Marianne Whitlock, Kennett Square, PA

ABSTRACT

The SAS® DATA step is easy enough for beginners to produce results quickly. You can use a model from a manual or a colleague's program, and adapt it to your problems. Typically, you code a "SET" or "INPUT" statement and proceed with assignment statements etc. to produce a new file from a previous one. You can easily print the results, or put the data into a statistical procedure, without ever troubling yourself with the Program Data Vector (PDV). When you want to accomplish something more complex though, you find that familiarity with the typical DATA step pattern is not enough, and if you have not internalized the concept of the PDV, you are not well equipped to tell SAS what you want.

Using increasingly complex problems, this talk will step through code from the PDV point of view. The goal will be to give confidence to try more manipulative programming. A set of rules governs the PDV, and at each point in the compiling and execution of the DATA step, it is possible to know exactly what information the PDV holds and what it will do with that information. Understanding this enables you to take control, and tell SAS what you want.

INTRODUCTION

The PDV, the holder of information while SAS is executing the DATA step, is the core of this talk. Related concepts will be covered, such as the rules SAS follows in changing the contents of the PDV, what SAS does in a MERGE, and pitfalls one can avoid by knowing the rules.

Example 1: Input and Output

A DATA step is processed in two phases - first it is compiled, and then executed. Let's look at a simple example. Consider the data set START:

G	Q
aa	5
aa	8
bb	3

Figure 1 - Data set START

and the following program. During the compile phase the program lines are read line by line. When the SET statement is read, the compiler will look at the data set START and determine its variables. The compiler reserves a place in the PDV for each variable as it is encountered. The PUT statements in the program show the PDV at various stages of the program.

```
data result ;
   put '1' _all_ ;
   set start;
   by G ;
   put '2' _all_ ;
```

```
      D = 2 * Q ;
      put '3' _all_ ;

      if last.G ;
      put '4' _all_ ;

   Run ;
```

The rows in Figure 2 show the values in the PDV each time one of the four PUT statements is executed. (The first row shows the variable names. The automatic variable _ERROR_ has been abbreviated as _E_ to fit in the table.)

Put #	G	Q	first.G	Last.G	D	_E_	_N_
1		.	1	1	.	0	1
2	aa	5	1	0	.	0	1
3	aa	5	1	0	10	0	1
1	aa	5	1	0	.	0	2
2	aa	8	0	1	.	0	2
3	aa	8	0	1	16	0	2
4	aa	8	0	1	16	0	2
1	aa	8	0	1	.	0	3
2	bb	3	1	1	.	0	3
3	bb	3	1	1	.6	0	3
4	bb	3	1	1	6	0	3
1	bb	3	1	1	.	0	4

Figure 2 - PDV at each PUT statement

Only one set of values (one row) at a time is available to SAS. but here we can see how the PDV changes during execution. The first PUT statement shows that the variables are all missing in the first line except for the automatic variables. Both FIRST.G and LAST.G, created by the use of a BY statement, are set to 1 in this first line by the compiler. In later lines they indicate the start (FIRST.G=1) and end (LAST.G=1) of each group. _ERROR_, for data errors, is initialized to 0. _N_ is initialized to 1.

The DATA step is an implied loop driven by input. The existence of an INPUT, SET, MERGE, or UPDATE statement, means, in effect, get a record. SAS gets values from an observation (a row) in a data set and puts the values in the PDV. It then proceeds line by line through the rest of the executable statements for that observation. Then it goes back to the top and goes through the whole DATA step for the next observation, and so on for every observation.

_N_ is incremented each time the implied loop starts again. Each line in the DATA step is executed in order, line by line, for each observation in the data set until the SET statement is to execute, and there are no more input records for the SET statement to read.

Take a closer look at the changes going on in the PDV as SAS executes each line. The second PUT statement is right after the SET statement, and shows values for G and Q obtained from the first observation in START. The variable, LAST.G, is now 0 because the value of G, 'aa', in this observation is not the last observation with the value, 'aa'. The box for D, a variable created in an assignment statement, and put in the PDV at compile time is still missing at this point.

After the assignment statement is executed, the third PUT statement shows that the PDV now holds the assigned value for D. The other values have not changed.

The next statement to execute is a subsetting IF. The most useful way to think of a subsetting IF is to think of the meaning of the rest of the statement which you don't have to type, that is: then continue executing the rest of the lines in the DATA step for this observation. Since last.G is 0, the condition is not passed, and SAS returns to the top of the DATA step for another iteration without executing any more statements. Hence there is no PUT #4 at this point. In this example, the only statement we see after the subsetting IF is the fourth PUT statement; however the significance is that each time the bottom of the DATA step is reached, if there is no OUTPUT statement anywhere else in the DATA step, SAS writes the current values of the PDV to the data set being created. So neither the fourth PUT statement, nor the default OUTPUT happens for the first observation because SAS does not continue executing the DATA step for the observation that does not pass the subsetting IF.

We have just completed one iteration of the implied loop of the DATA step. SAS returns to the top of the DATA step, increments _N_, and executes the first PUT statement again. Notice that D is missing again because SAS by default sets variables from ASSIGNMENT statements and INPUT statements to missing at the top of the DATA step. In contrast, the values of variables coming from SET statements are retained until they are replaced. This is why G and Q still have the values they obtained from the first observation.

In this second iteration of the DATA step loop, SAS reads the last observation of the BY group with G='aa'; the subsetting IF test is passed and the fourth PUT statement is executed. This observation is written to the data set RESULT because we are at the bottom of the DATA step and there is no explicit OUTPUT statement in the code.

The above notes are somewhat long because a lot of things are going on in even this very simple example.

Now, let's skip over the four PUT statements of the last observation (all of which are executed), and look at the values in the PDV shown in the last line of Fig 2. Although SAS has read the last observation and executed every line in the DATA step, the looping does not stop. SAS goes back to the top of the DATA step, increments _N_ to 4, and executes the first PUT statement. There are no more results of PUT statements because the SET statement causes the DATA step to stop when there are no more records to read.

Before trying to do something a little more manipulative in a DATA step, let's review some important concepts illustrated in the first example:

1) The DATA step is an implied loop (when an INPUT, SET, MERGE, UPDATE or MODIFY statement is present).
2) SAS executes the statements in order, line by line, for each observation in the data set being read.
3) _N_ counts iterations of the implied loop of the DATA step.
4) At the bottom of the DATA step, SAS, by default, writes an observation to a SAS data set.

Example 2: Restructuring a data set

Let's look at an external file, in Fig. 3, with data from a questionnaire about activities: sports and hobbies. The first column has each respondent's identifier. The second has 'H' for hobby or 'S' for sport. The other columns hold the responses regarding hobbies and sports. Each record for a respondent can hold up to three sports or hobbies. A respondent may have several sport or hobby records or even none of a given type. The data in Figure 3 do not show all of these possibilities because we must keep the data extremely simple in these examples.

Say we want counts of how many times a particular response was given for a sport or hobby. For example "stamps" was given as an answer twice. We want to create a data set, ACTIVITY, of sports and hobbies, so we can easily do the counting task. Now the DATA step involves restructuring the input data of Fig. 3 to that in Fig. 4.

1	H	photo	Stamps	
1	S	baseball		
2	H	stamps	Travel	photo
2	S	hockey	Golf	tennis

Figure 3 - Input data

OBS	ID	TYPE	ACT
1	1	H	photo
2	1	H	stamps
3	1	S	baseball
4	2	H	stamps
5	2	H	travel
6	2	H	photo
7	2	S	hockey
8	2	S	golf
9	2	S	tennis

Figure. 4 - Output SAS data set

The following DATA step does the restructuring required for a PROC FREQ. In addition, it has 4 PUT statements just to show the PDV at four points of interest.

```
data activity(keep=id type activity);
  length act $ 8 ;
  infile activity missover ;
  put '1: ' _all_;
  input id $ type $ @ ;
  put '2: ' _all_;
  do until (activity=' ') ;
    put _'3: ' all_;
    input activity $ @ ;
    put '4: ' _all_;
    if activity ne ' ' then output;
  end;
 run;
```

Again, if we look at the PDV as the DATA step executes, we can see how the statements achieved the changing of the data structure of the external file to the data structure of the SAS data set. Each row of the table in Figure 5 shows the PDV at the time the PUT statements execute. The results for ID=1 are shown together with the PDV when the DATA step stops. What is of interest is how the two records for ID #1 in the input data set become three records in the output data set. We will see this through the changing (and sometimes unchanging) values in the PDV.

#	ID	TYP	ACTIV.	_E_	_N_
1	.			0	1
2	1	H		0	1
3	1	H		0	1
4	1	H	photo	0	1
3	1	H	photo	0	1
4	1	H	stamps	0	1

3	1	H	stamps	0	1
4	1	H		0	1
1	.			0	2
2	1	S		0	2
3	1	S		0	2
4	1	S	baseball	0	2
3	1	S	baseball	0	2
4	1	S		0	2
	...	...	...	...	...
1	.			0	5

Figure 5 - PDV at each PUT statement

The first column in Figure 5 is not part of the PDV, but shows which view of the PDV, the result of the four PUT statements, we are seeing. The variable names are abbreviated in order to fit the table in the column.

The first row in Figure 5 shows the PDV before the first INPUT statement (#1). ID, TYPE, and ACTIVITY all have missing values since their values come from INPUT statements. The automatic variables, _ERROR_ and _N_ were initialized at compile time to 0 and 1 respectively.

The first INPUT statement reads only the first two fields, ID and TYPE, because values for ACTIVITY will be read in a loop. Hence row #2 shows the PDV with its values for ID and TYPE.

The trailing @ on the INPUT statement tells SAS to hold the line of the file being read, not move on to the next line, as an ordinary INPUT statement would do. We want to hold the line in order to read the rest of the data on the same line in an explicit loop. An explicit OUTPUT statement is needed in this loop to write an observation for every value found. Had we used the default output activity, there would be only one record for each activity type.

The third PUT statement, at the top of the explicit DO loop, shows no change to the PDV since further input has not yet taken place. The fourth PUT shows the result of the second INPUT statement. Now ACTIVITY has the value "photo".

The second INPUT statement also has a trailing @, so that SAS will again hold that same line for the next input, and keep doing so. The line is eventually released when the explicit DO-loop ends and we are at the bottom of the DATA step going to the next iteration of the implied DATA step loop.

The test for ACTIVITY passes since it has the value "photo". Thus the OUTPUT statement is executed causing an observation to be written to the output SAS data set. ID, TYPE, and ACTIVITY from the current contents of the PDV are written because of the KEEP= option on the output data set. At this point we have observation #1 in Figure 4.

Note that 1) SAS has not yet finished with the first record of the INPUT data set since the line has been held by the trailing @, and 2) we are at the bottom of the DO-loop.

Now the value of ACTIVITY is tested and found not missing (still "photo"). So we go through a second iteration of the DO-loop. PUT #3 comes again with ACTIVITY still having the value "photo".

The second INPUT statement now reads the next activity on the first record because the line was held by the trailing @. After the input we come to PUT #4 again. This time the value of ACTIVITY has changed to "stamps".

Output takes place again because ACTIVITY is not missing. Notice that the value 'H' for TYPE, which was read before the DO loop, is still in the PDV because the PDV does not get cleared by an OUTPUT statement. Consequently, TYPE is 'H' on the second observation.

We are at the bottom of the DO-loop for the second time. ACTIVITY is "stamps" so the test is passed and

we begin a third iteration of the DO-loop. The first line out input is read again. This time ACTIVITY is blank as shown by PUT #4. Consequently output is not performed and the DO-loop stops.
We are at the bottom of the DATA step for the first time as can be seen by looking at _N_ in the row for the last PUT (#4). The first input record is now released because a single trailing @ was last used on an INPUT statement. (Had a double trailing @ been used the line would still not have been released.)

Remember that the DATA step *itself* is an implied loop, and input was done so SAS goes back to the top of the DATA step, increments _N_ to 2, and starts executing the code for the second record.

This time the DO-loop has only 2 iterations, with only one OUTPUT, and we now have all the information for ID=1 in a new form in the SAS data set.

In our example we read two input records during two iterations of the implied DATA step loop and wrote 3 observations for ID = 1. The remaining records are read in a similar fashion with appropriate output being performed in the DO-loop. The discussion of this activity is skipped and the PDV is shown by "**...**" because there is nothing new here.

After the last output observation is written, we continue with the details. We are at the bottom of the DATA step, and input has been done, so we have one more iteration of the DATA step loop. As you can see in the last row of Figure 5, _N_ is now 5 (we read 4 records in 4 iterations of the DATA step loop). The last row came from PUT #1 because the next line to execute is the first INPUT statement and we are at the end of file. Consequently the DATA step stops executing at this point.

In summary, the concepts that the PDV helped to illustrate in example 2 are:

1) When external data are restructured one may not have the common DATA step structure of one record input and observation output.

2) A SAS DATA step is an implied loop, as was mentioned in example 1, but we can code our own iterative loops within the SAS loop.

3) Variables are not set to missing during the DATA step, only at the top for variables named in INPUT and assignment statements. Therefore the values of TYPE, S and H, remained in the PDV for writing out repeatedly to new observations.

The PUT statements in the preceding example caused information about the PDV to be written to the log at key points. Another method of checking on what's going on in the PDV is the use of the DATA step debugger. With this facility, you can step through the statements asking to see the PDV any time you touch a key programmed with the command DSD EXAMINE _ALL_. This method will be demonstrated as part of the talk.

Example 3: Data set created at end of file

In this example there is no correspondence between input observations and output observations, since the entire output file is written after all input observations have been read.

Say we have a data set showing the date a package was sent and the date it was received:

PKG_ID	SENT	RECEIVED
BK1	27AUG05	31AUG05
HK7	31JUL05	06AUG05
LT6	03AUG05	05AUG05
MT9	09JUL05	16JUL05
RL3	01SEP05	04SEP05
TJ4	05AUG05	10AUG05

Figure 6 - SAS data set PACKAGES

We want a report showing how many packages fall into different groups based on how many days passed between the sending and receiving of the package. The number of days is grouped as shown in Figure 7.

Number of days	Count of packages
< 3	1
3-5	3
> 5	2

Figure 7 - SAS output data set REPORT

The DATA step to handle this task illustrates several new ideas:

1. The use of a DATA step subroutine
2. Testing for end of file
3. Temporary arrays
4. Number of output observations have no correspondence to the number of input observations

As we read the input data set, we want to accumulate the information for the report. We know we want the output data set to have the two variables in the report, so let's list them in a KEEP= option on the DATA statement. We will not keep any of the variables in the input data sets, nor any of the working variables used during the DATA step to accumulate information. The only variables kept are the ones created in the routine done at end of file. Here is the program:

```
data report(keep=num_days pkgcount) ;
   *** 1) top of DATA step ;
   array label (3) $ 3 _temporary_
         ('< 3','3-5','> 5') ;
   array count (3) _temporary_ ;

   if eof then link report;

   set packages end = eof ;

   diff = received - sent ;
   if diff < 3 then i = 1 ;
   else
   if (3 <= diff <= 5) then i = 2 ;
   else if diff >= 6 then i = 3 ;
   count (i) + 1 ;
   *** 2) bottom of DATA step ;
return;

report:
   do i = 1 to 3;
      *** 3) top of DO-loop ;
      num_days = label (i) ;
      pkgcount = count(i) ;
      output ;
      *** 4) bottom of DO-LOOP ;
   end;
return;
run;
```

In this case the bottom of the DATA step where output is normally done by default occurs at the first

RETURN statement because "bottom of the DATA step" refers to the last executed line, not the last physical line. But no default output takes place here because the OUTPUT statement appears in the REPORT subroutine.

Technically the PDV has these variables:

EOF PKG_ID SENT RECEIVED DIFF I NUM_DAYS PKGCOUNT

in that order because that is the order the compiler encounters them.

TEMPORARY arrays are stored in contiguous memory (for fast access) in a different area of the executable object module. However we will consider them as part of an extended PDV (XPDV).

There are four points of interest in the DATA step we might want to consider as given by the numbered comments. In the previous examples we used PUT statements to capture the PDV, but it is time to realize the importance of the PDV as a tool of thought, so we have done away with the real PUT statements, and left comments in the program at the points of interest.

Now the XPDV in this example is too long to picture it as we previously did. Instead let's look at the variables of interest in a first column with the next columns, numbered to correspond to the comment numbers in the program, and showing the values the PDV holds for those variables at those points. For example, Figure 8 shows the set of PDV values for _N_ = 1. Recall #1 was at the top of the DATA step and #2 at the bottom.

VAR	#1	#2
PKG_ID	**.**	BK1
SENT	**.**	27AUG05
RECEIVED	**.**	31AUG05
EOF	0	0
DIFF	**.**	4
I	**.**	3
NUM_DAYS		
PKGCOUNT	**.**	**.**
ERROR	0	0
_N_	1	1
COUNT1	**.**	**.**
COUNT2	**.**	**1**
COUNT3	**.**	.

Figure 8 - XPDV during iteration 1

Now let's skip to the reading of the last observation when _N_ is 6, shown in Figure 9.

VAR	#1	#2
PKG_ID	RL3	TJ4
SENT	01SEP05	27AUG05
RECEIVED	04SEP05	31AUG05
EOF	0	1
DIFF	**.**	5
I	**.**	2
NUM_DAYS	**.**	**.**
PKGCOUNT	**.**	**.**
ERROR	0	0

_N_	6	6
COUNT1	1	1
COUNT2	2	3
COUNT3	2.	2

Figure 9 - XPDV during iteration 6

#1, the top of the DATA step for iteration 6, is *before* the SET statement. It is still holding values for PKG_ID, SENT, and RECEIVED from the previous observation (RL3, 01SEP05, and 04SEP05). DIFF, created by an assignment statement has been set to missing. COUNT1-COUNT3 are retained because temporary arrays are retained. Even if not part of a temporary array, the use of the sum statement causes SAS to retain values. When SAS retains, we can accumulate information through all the iterations and have totals available when SAS is finished processing the whole file.

#2, the bottom of the DATA step for iteration 6, shows the last observation on the file, the three COUNT variables holding the full information for the report, and EOF =1.

It is significant and very useful to know that SAS does not then stop the DATA step. Whenever there is input as part of the DATA step, SAS iterates until it comes to INPUT, SET, MERGE, or UPDATE statement with no more records to read. Therefore, in this example, SAS would go to the top of the DATA step, increment _N_ to 7 and execute statements until the SET statement.

The first executable statement is the one that tells SAS to link to the subroutine REPORT. The information we need for the report has been accumulated in COUNT1-COUNT3, and the descriptive information for these counts is in another temporary array, LABEL. All of the statements in the sub-routine will be executed even though SAS has finished reading the file. The subroutine takes information, which is in one dimension, the PDV, and transposes it, through OUTPUT three times within a DO loop to columns of information as shown in figure 10.

For the last iteration of the implied loop, iteration 7, we use the abbreviations a, b, and c to refer to the labels '< 3', '3-5', and '> 5' to allow Fig. 11 to fit in the column.

VAR	#1	#3	#4	#3	#4	#3	#4
EOF	1	1	1	1	1	1	1
NUM_DAYS	.	.	a	a	b	b	c
PKGCOUNT	.	.	1	1	3	3	2
I	.	1	1	2	2	3	3
ERROR	0	0	0	0	0	0	0
_N_	7	7	7	7	7	7	7
COUNT1	1	1	1	1	1	1	1
COUNT2	2	3	3	3	3	3	3
COUNT3	2	2	2	2	2	2	2

Fig. 10 - XPDV during iteration 7

Each time SAS gets to #4 above it has just done an OUTPUT. Since the output data set contains only NUM_DAYS and PKGCOUNT, the values for those variables in the three #4's above are written to REPORT.

When SAS has executed all the statements in the subroutine, the RETURN statement tells it to go back to the line immediately after the LINK statement and continue. The next statement is the SET statement, there are no more records, and SAS stops at this point.

It may seem more intuitive to test for end of file at the bottom of the DATA step, but once you know how SAS works, you will understand what a good habit it is, whenever your DATA step has a test for end of file, to code the test above the SET statement. The reason is: A DELETE or subsetting IF (a DELETE stated positively) tells SAS not to execute the rest of the statements in the DATA step for the observation being deleted, and instead return to the top of the DATA step for the next iteration. If the last record on the input file is one that gets deleted, SAS will not execute any more code after the DELETE for that observation, so the test for end of file would never be reached. If there is no DELETE in the DATA step, the habit can't hurt, and may be protective in case someone modifies the code to put a DELETE in it.

The main purpose of example 3 was to encourage adventurousness, creativity, and control. When you visualize the PDV and what information SAS is holding in it, and the rules it follows, you can take control and manipulate the information as you wish. Input and output looked nothing like each other, but the information for the output was built and maintained in the PDV.

Concepts illustrated in example 3 were arrays, indexes to arrays, temporary arrays, sum statements, retaining v. setting to missing, testing for end of file, and sub-routines using the LINK statement.

Example 4: MERGE and more on RETAIN

We want to look up a RATE for each ID in data set A (on the left in figure 11). Data set B (on the right) has a base rate for each ID in A.

Data Set A

ID	DAY	QUANTITY
M	1	5
M	2	2
M	3	3
R	1	4
R	2	6

Data Set B

ID	RATE
M	2
R	3

Figure 11

The RATE is to be multiplied by the QUANTITY to get COST, but we want to subtract 1 from the RATE when the QUANTITY is greater than 4. Since that is true for ID=M, DAY=1, and ID=R, DAY=2, what we *want* is to lessen the RATE found in data set B for these two cases and obtain the results shown in data set C (wanted) in figure 12.

Data set C (wanted)

ID	DAY	QUANTITY	RATE	COST
M	1	5	1	5
M	2	2	2	4
M	3	3	2	6
R	1	4	3	12
R	2	6	2	12

Figure 12

So we write the code:

```
data c ;
   merge A
         B
   ;
   by ID ;
```

```
   if quantity > 4 then rate = rate - 1;
   cost = rate * quantity;
run;
```

Instead of what was wanted, the code produces the data set C in figure 13.

Data set C (result of code)

ID	DAY	QUANTITY	RATE	COST
M	1	5	1	5
M	2	2	1	2
M	3	3	1	3
R	1	4	3	12
R	2	6	2	12

Figure 13

Did SAS forget the RATE we looked up in Data Set B for ID=M? Yes, it did. By using the same variable to store the reduced rate, we told SAS to forget the rate we looked up. That was not our intention, so we will have to take control by using the way *SAS* works to get the results *we* want.

Let's look at the PDV right after the MERGE statement has been executed for the first iteration of the DATA step (_N_=1). In a MERGE, SAS will look for the lowest value of the BY variable in each data set to read into the PDV first, or, in the case of a match, read an observation from both data sets at the same time. ID=M from data set A matches ID=M from data set B, so the first observation from A and the first observation from B come into the PDV:

ID	DAY	QUANTITY	RATE	COST
M	1	5	2	.

Figure 14

The next two executable statements cause the RATE to be changed from 2 to 1, and the COST to be calculated (1*5=5). We are then at the bottom of the DATA step and a record is written to Data set C with the values as shown in Figure 15.

ID	DAY	QUANTITY	RATE	COST
M	1	5	1	5

Figure 15

A rule of SAS is that variables coming from a data set in a SET, MERGE, or UPDATE statement have values retained in the PDV over an entire BY group, until something causes them to be replaced. We will see the result of this.

Let's move to iteration 2 (_N_=2). Data set A has another observation with the same value for ID (M) as in the former iteration, but Data set B has no more observations with ID=M. SAS brings the second observation from Data set A into the PDV. SAS brings an observation in only once, and since the observation with ID=M from Data set B has already been brought in, SAS does not bring it in again. Variables coming from a SET, MERGE, or UPDATE statement are retained though, so the RATE=1 that went into the PDV at _N_=1 is still there, not the RATE=2 of the "look-up" data set, Data set B. The RATE of 1 continues to be retained for the rest of the BY-group (values of M for ID), and is set to missing at the beginning of the next BY group (ID=R).

Here is one way to get what we want:

```
data c (keep=id day quantity rate cost) ;
  merge A
        B (rename=(rate=lkuprate))
```

```
   ;
  by ID ;

  if quantity > 4 then rate=lkuprate - 1;
  else rate = lkuprate ;
  cost = rate * quantity;
run;
```

In effect we keep the value for RATE from the Data set B, by storing it in the PDV under LKUPRATE. We then do our look-up for every observation in A in LKUPRATE, and then use it or change it when creating RATE to be used in the multiplication.

Example 4 is unusual. The most common thing wanted, when merging a data set with multiple values of the BY variable with a data set with unique values of the BY variable, is to retain values from other variables in the unique data set over all the observations multiple data set. Wanting to change the value being retained while still in the same BY group, and then wanting to go back to the original value is the exception. Exceptions come along often, and if we know how SAS works, we will be in control to get what we want.

Example 5: MERGING to correct values

Example 4 illustrated how to modify values from a look-up data set in a MERGE. Another related problem is how to merge a data set that needs fixing with a data set of corrections. In this case the same variable exists on both the "bad" data set and the "good" one. You may hear the advice to place the "good" data set on the right in the MERGE statement because values from the left data set will be overwritten with values from the right one. Be wary of this advice. We have seen in example 4 that things are not so simple. When the "left" data set (the one with the "bad" values for some variable) has multiple records in the BY group, and the "right" data set has unique records in the BY group. The first observation in the BY group is corrected, but all the others remain uncorrected because the variable to be fixed has only one location in the PDV. On the first observation of the BY group it first achieved the "bad" value and then the "good" one as expected. But on the second observation the next "bad" value replaced the "good" and this time there is no second "good" value so the "bad" value remains.

A basic rule to follow in a MERGE is never allow the same variable name in both data sets, except BY variables. To solve the problem, drop the variable with the "bad" values, or if, under some conditions, you want them, rename the variable from the "good" set. Now the "good" variable will be retained and available for use whenever necessary. Notice that now the distinction of the left data set and the right one is no longer of importance.

How can one insure that no overwriting is done? Use KEEP= options on the data sets involved, and check to make sure there are no common variables except the BY variable(s). Here is an example where the use of KEEP= may be important.

```
data new ;
 merge geog (keep=id type state city)
       demog (keep=id type eth inc )
       newinfo (keep=id type pop)
  by id type ;
  other statements
run;
```

The KEEP= option is worth the time and effort it takes to write it. It will drop unwanted variables, add clarity, increase efficiency, and provide internal documentation.. An extra is that you will not have to be nervous about merging more than two data sets at a time, because the KEEP= on each data set will remind you of what is going on.

On the other hand, there *is* a reason to be concerned with the order you list data sets in a merge. If your BY variables (the only variables in common) have different attributes, SAS will use the attributes of the variables in the *first* named data set, not the *last* named data set. For example in:

```
merge a b ;
by x ;
```

the attributes length, label, and format are determined by the attributes of X on data set A. Remember the PDV is set up at compile time, and SAS determines attributes of variables at the first place it encounters them at that time (with certain exceptions, such as KEEP= on output data sets, RETAIN statements without initialization, and LABEL statements.

One more general rule for merges is that whether you are merging two or 50 data sets, you should have a maximum of one of the data sets with multiple observations with the same value of the BY variable. A multiple to multiple merge does not make sense. An observation in a data set needs some way of being identifiable other than its order. If you do a multiple to multiple merge, you would be relying on order, and that weakness would probably wreak havoc when any modification to the program is made.

Other PDV issues:

No discussion of the PDV would be complete without mentioning the importance of distinguishing between non-executable (compile time) and executable statements in SAS DATA step processing. In example 1 we saw that SAS set up the PDV during compile time, and determined the order the variables would be in on any output data sets. In example 5, we saw that SAS also determined the attributes of the variables at compile time.

At compile time SAS determined whether a variable would be retained or set to missing at the beginning of each iteration of the DATA step. ATTRIBUTE, LENGTH, FORMAT, ARRAY KEEP, DROP, and RETAIN are non-executable statements. SAS reads them and uses the information at compile time; they are not even part of the execution module. Knowing this can help the programmer to understand certain actions better. Take the RETAIN statement. SAS allows the user to have a variable retained and initialized in the same statement. An example is:

```
RETAIN X 1 ;
```

When you use the RETAIN statement this way, you are telling SAS to initialize X with the value of 1, and not set it to missing at the top of the DATA step. Those are two separate instructions, which SAS lets you put in one statement. However the value of X can be changed at any time by another statement. The RETAIN only tells SAS not to set X to missing at the beginning of each iteration. If you realize that RETAIN is non-executable, you won't see X being set to 1 again in every iteration.

CONCLUSION:

Visualizing the PDV is a very helpful way to increase your understanding of what is going on in DATA step processing. It may not seem necessary for very simple DATA steps, but even then, it can help. If you visualize on some level, a record being read in and a record being output for every observation in a data set, you will be less likely to write minuscule DATA steps like:

```
data a;
   set perm.a ;
data b;
   set a ;
   x = y + 2;
```

As long as each observation of PERM.A is to be copied into the PDV, why not create X while it is there?

When you want to do something more complex, you need concrete, not vague, knowledge of the information the PDV is holding, and the rules SAS observes in manipulating that information. When your program is not doing what you intend, use PUT statements to write parts of the PDV to the log, or use the DATA step debugger to show you what is going on in the PDV.

The author can be contacted by mail or e-mail:

Marianne Whitlock
29 Lonsdale Lane
Kennett Square, PA 19348

mariannewhitlock@Comcast.net