ABSTRACT

The RETAIN statement is a tool that is used to do such tasks as LOCF. However, it becomes clumsy when processing BY group variables. When the boundary of a BY group variable is crossed such as the next patient, care must be taken while using RETAIN to prevent the carrying over of the previous patient’s value. This paper examines the RETAIN statement in detail and explores alternative approaches when certain conditions are met. The paper also devotes a section to LAG as this seems to be a corollary to the RETAIN statement.

INTRODUCTION

The SAS® documentation has devoted a significant section to the RETAIN statement explaining many finer points. Therefore, it behooves us to explore it and understand it well. In addition we will cover other aspects relevant to RETAIN that are perhaps not as well documented.

After we have examined the RETAIN statement in depth, we will explore alternatives to achieving the same functionality and discuss some of the merits and drawbacks of the alternative method.

We shall also devote a section to LAG as there are some similarities, especially, in the context of LOCF in the Pharmaceutical industry.

All participants from beginner to advanced are encouraged to attend. However, it does require a good understanding of the data step to get the full value from the presentation. In any case, beginners are encouraged to read the paper as it covers material that is applicable to the Pharmaceutical business.

RETAIN DEFINED

Let us start with how the RETAIN statement is defined in the Online documentation.

Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next (SAS Institute Inc., 2004).

There are two key aspects to this definition.

1. RETAIN is applicable ONLY to new variables created in the data step.
2. Values are retained across ITERATIONS of the data step.

The first point makes it clear that variables that come from an input data set through a SET, MERGE, MODIFY or UPDATE statement are automatically retained. Naturally, all automatic SAS variables such as _N_, _IORC_ are also automatically retained from one iteration of the data step to the next. The second point, above, will be the focus of the discussion when we examine changing the default behavior of the data step.

We will now begin our focus on RETAIN by walking through some examples of its different usages.

FINER DETAILS OF RETAIN EXPOSED

Let us look at some of the ways a RETAIN statement can be used. The documentation explains many different usages and not all of them have examples. Let us look at them one by one with simple examples.

1. Initializing a Constant

First let us examine how to initialize a variable’s value and carry it through the data set after creating some test data.
data test ;
  input x ;
cards ;
1 2 3
run ;

options nocenter ;
proc print ;
run ;

This gives us:

Obs  x
1   1
2   2
3   3

Let us assume that we want to assign the value of 9 to a constant C and carry it across all the observations. It is easy to do this with a RETAIN.

data test2 ;
  set test ;
    RETAIN C 9 ;
run ;

proc print ;
run ;

This produces:

Obs  x  C
1   1  9
2   2  9
3   3  9

A character value can also be initialized this way but the value must be quoted. An example would be:

data test3 ;
  set test ;
    RETAIN CHAR "Treatment" ;
run ;

Obs  x      CHAR
1   1    Treatment
2   2    Treatment
3   3    Treatment

(2) RETAIN ALL NEW VARIABLES

You may be tempted to think that the wonderful _ALL_ is the answer to this. It can be, but only under certain strict conditions. The same applies to the special lists _CHAR_ and _NUMERIC_. If you use these special lists make sure all the variables that you want to RETAIN are already defined. To understand the implications of not defining prior to the RETAIN, run the following code and carefully examine the consequences.
data test4 ;
set test ;
array a (3) ;
RETAIN _ALL_ ;
if _n_ = 1 then do ;
do i = 1 to 3 ;
a (i) = (i) ;
end ;
b = 7 ;
end ;
run ;

What happens to the variable B? The first observation is assigned a value of 7 and then it is reset to missing at the next iteration of the data step. Why? The RETAIN _ALL_ construct works this way. The _ALL_ refers to all the variables that have ALREADY been defined. In this case the variable B is defined AFTER the RETAIN statement. This begs the question “how do we achieve this functionality of retaining every single variable that is newly created?” The answer is much simpler than you think.

data test5 ;
set test ;
array a (3) ;
RETAIN ;
if _n_ = 1 then do ;
do i = 1 to 3 ;
a (i) = (i) ;
end ;
b = 7 ;
end ;
run ;

A simple retain with nothing next to it achieves this. However, such a construct might have limited practical application.

(3) INITIALIZING A LIST OF VARIABLES

This is a list of variables NOT comprising an array. There are at least two ways to do this. The first is a naturally corollary of (1) above. The second is simply a different representation.

data _null_ ;
retain a 1 b 2 c 3 ;
put (_all_) (=) ;
run ;
data _null_ ;
retain a b c (1 2 3) ;
put (_all_) (=) ;
run ;

Both these data steps write the following results to the log:

a=1 b=2 c=3

If you want a constant value for all of the variables in the RETAIN statement you can also use an iterator value as below:

data _null_ ;
retain a b c (3*1) ;
put (_all_) (=) ;
run;
This particular construct of the PUT statement may be unfamiliar to the reader. This is a named put list that is formatted to display with an Equal sign. The advantage of this versus simply using PUT _ALL_ is that we do not get the dump of _N_ and _ERROR_ in the log.

(4) UNINITIALIZED VARIABLE FROM A RETAIN

When you specify a variable name in a RETAIN statement without initializing and then do not reference it later with an ASSIGNMENT statement then that variable is NOT WRITTEN to the output data set.

data uninitialized;
   retain A;
   do B = 1 to 2;
      output;
   end;
run;

A note is issued in the log:

NOTE: Variable A is uninitialized.
NOTE: The data set WORK.UNINITIALIZED has 2 observations and 1 variables.

However, if the variable A is referenced in any other manner it is WRITTEN out but is still uninitialized. That is, if another variable is assigned with a value of A. Notice that all three variables are written out to the data set in the log for the following data step:

data q;
   retain A;
   do B = 1 to 2;
      c = a;
      output;
   end;
run;

NOTE: Variable A is uninitialized.
NOTE: The data set WORK.Q has 2 observations and 3 variables.

(5) THE SPECIAL CASE OF ARRAYS

Arrays and RETAIN have an interesting relationship. Temporary arrays by definition are automatically retained while arrays defined for creating new variables are not. However, when the latter array is initialized with values then those variables are retained as well. Let us examine these scenarios with examples.

Here is a data step log that shows that newly created variables are not RETAI Ned.

270   data _null_;  
271      set test;  
272      array a(2);  
273      if _n_ = 1 then a2 = 9;  
274      put a(*)=;  
275   run;

a1=. a2=9
a1=. a2=.
a1=. a2=.  

Here is a data step log that shows that even partially initialized arrays carry a RETAIN on all the elements of the array.

233   data _null_;
set test ;
array a(2) (1) ;
WARNING: Partial value initialization of the array a.
if _n_ = 1 then a2 = 9 ;
put a(*)= ;
run ;
a1=1 a2=9
a1=1 a2=9
a1=1 a2=9

Note that even though the array is only partly initialized it still applies a RETAIN to all the elements of the array. That is A2 is also RETAINed.

An array can also be explicitly retained by simply specifying the name of the array in the RETAIN statement.

data _null_ ;
set test ;
array a(2) ;
retain a 9 ;
put a(*)= ;
run ;
a1=9 a2=9
a1=9 a2=9
a1=9 a2=9

The TEMPORARY array is difficult to demonstrate directly. However, you could either use the data step debugger to view the automatic retain or you could assign their values to real variables and view them. This is left to the reader as an exercise but the code to use in the data step debugger is below:

data q / debug;
  set test ;
  array a(2) _temporary_ ;
  if _n_ = 1 then do ;
    a(_n_+1) = 2 ;
  end ;
run ;

The point is that TEMPORARY arrays create temporary variables which are automatically retained like all SAS created automatic variables.

(6) THE ACCUMULATOR VARIABLE IN A SUM STATEMENT

You are probably familiar with the following construct

VAR1 + 1 ;

This forces a RETAIN on VAR1 and it need not be specified explicitly. Here is a simple illustration.

data _null_ ;
set test ;
if _n_ = 1 then y + x ;
put x = y = ;
run ;
x=1 y=1
x=2 y=1
x=3 y=1
As can be seen, the value of Y persists after the first iteration even though there is no explicit RETAIN.

**(7) THE MAGIC OF SOME AUTOMATIC VARIABLES**

There is a section in the RETAIN documentation titled REDUNDNACY. This mainly addresses some of the issues covered above that have an automatic RETAIN. In addition it can be inferred that most SAS created automatic variables can be initialized with a RETAIN and assigned to another variable to take advantage of the automatic retain. Here is an example with the iterator that an implicit array uses.

```sas
341  data _null_ ;
342    set test ;
343    retain _i_ 9 ;
344    y = _i_ ;
345    put x= y= ;
346  run ;
```

x=1 y=9
x=2 y=9
x=3 y=9

Although this example is trivial, it illustrates the point that SAS created automatic variables can be used as long as we know the rules the data step follows. Efficiency can be achieved using automatic variables that need not be explicitly dropped after a data step. In big data sets the difference in the processing speed can be impressive.

**A PHARMACEUTICAL APPLICATION OF THE RETAIN**

A common task in Clinical trials is the imputation of missing values. The bulk of the summary tables use the LOCF or The Last Observation Carried Forward method of imputation. To facilitate this, a RETAIN is often used. Let us assume we are working with Systolic blood pressure for a single patient and these are the patient’s readings in 4 visits.

```sas
data systolic ;
  input sbp ;
  cards ;
  120
  160
  .
  140
run ;
```

Our task is to do an LOCF imputation for the missing value.

```sas
data impute ;
  set systolic ;
  retain newsbp ;
  if not missing(sbp) then newsbp = sbp ;
run ;

proc print ;
run ;
```

This yields:

```
Obs  sbp   newsbp  
1    120   120   
2    160   160   
3      .    160   
4    140   140   
```
Our objective is achieved, but how practical is the assumption of having only a single patient? Let us try this again with 2 patients and notice how clumsy the RETAIN becomes.

```sas
data impute2 ;        (1)
    set systolic2 ;     (2)
    by ptno ;          (3)
    retain newsbp ;    (4)
    if first.ptno then newsbp = . ;  (5)
    if not missing(sbp) then newsbp = sbp ;  (6)
run ;

proc print ;
run ;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>ptno</th>
<th>sbp</th>
<th>newsbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>.</td>
<td>160</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>.</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>115</td>
<td>115</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>108</td>
<td>108</td>
</tr>
</tbody>
</table>

This does give us our desired result. However, the syntax had to account for the carryover of values from the previous patient. This is where care must be exercised when using the RETAIN statement. Let us dissect the above data step and understand each step of the code.

1. This is the instruction to create data set IMPUTE2.
2. Read one observation at a time from SYSTOLIC2.
3. Read all the observations for one patient before going to the next patient.
4. Do not clear NEWSBP’s value at the next iteration of the data step.
5. Set NEWSBP’s value to missing when a new patient’s first record is read.
6. Assign only non-missing SBP values to NEWSBP.
7. The RUN statement marks the data step’s boundary.

When the RUN statement is encountered, the data step control returns to the top to begin a new iteration. Coincidentally every new iteration also reads the next observation from the input data set. When a new iteration is begun, the data step clears the Program Data Vector (PDV) of the values of all NON-RETAINed variables and sets them to missing. However, if there is either an implicit RETAIN or an explicit RETAIN the values are carried over from the previous iteration. Since the data step cannot discriminate between patients it carries over values of NEWSBP from the previous patient when the patient boundary is crossed. Hence the clumsy construct of resetting it to missing at FIRST.PTNO.

**DO WE REALLY NEED THAT UGLY RETAIN FOR LOCF?**

The proposed alternative is to use a special do loop called the DOW. It is named after its author Ian Whitlock who first proposed it on SAS-L and was immediately popularized by Paul Dorfman who is another SAS-L great. For a more detailed explanation of the origins of the DOW, please refer to a previous SUGI and a PharmaSUG paper by this author. However, we will cover the bare minimum here to have an understanding of how this works. Here is the syntax that achieves the same result as the retain:

```sas
data impute3 ;
do until (last.ptno) ;
    set systolic2 ;
    by ptno ;
    if not missing(sbp) then newsbp = sbp ;
run ;
```
Let us spend some time understanding how this is different from the more traditional data step construct that preceded this. Notice that there is NO RETAIN and NO ugly resetting of values for the next patient.

The DO UNTIL loop reads one observation per iteration (of the loop and not the data step iteration) from a single patient. The LOOP never touches the bottom or return to the top of the data step until the last observation for that patient is exhausted. Since the loop does not reach the bottom every observation that is read is not automatically output. This is the reason that an explicit OUTPUT statement is required to write the observation to IMPUTE3. Since the data step does not begin a new iteration until all the observations for a patient are exhausted, it also does not clear the value of any newly assigned variable until the next iteration of the data step. When the last observation for a patient is read and the DO LOOP tests for LAST.PTNO to be true, the data step returns to the top to begin a new iteration. Now the automatic clean-up of NEWSBP occurs and it is set to missing before the next patient’s observations are read. Recall that in the default data step this was explicitly set to missing for FIRST.PTNO.

The code is a significant improvement over the traditional data step using the RETAIN. Efficiency has been achieved by eliminating the RETAIN and the need to reset the value to missing.

That said, the typical Pharmaceutical programmer is used to the default behavior of the data step and might not be easily convinced. Keep in mind that SAS documentation exists for the above data step but not in such explicit detail and certainly not in a direct way. However, every single part of the code can be backed with appropriate documentation from SAS. Other than these drawbacks, if one takes the time to understand how it works it can be immensely beneficial in other contexts as well.

WHERE DOES LAG COME INTO THE PICTURE?

Recall that we covered a section called REDUNDANCY under RETAIN. It is redundant to name the following variables in a RETAIN statement that are read with a SET, MERGE, UPDATE or MODIFY. This is because they are automatically retained. Taking the simple example of SET, is it possible that a LAG can be achieved? It is not as far fetched as one might think.

```sas
data _null_; 
  previous_x = x ;
set test ;
put (_all_) (=) ;
run ;
```

previous_x=. x=1
previous_x=1 x=2
previous_x=2 x=3

Taking advantage of the automatic RETAIN we capture the value of X that was RETAINed from the previous observation before the next value of X is read. However, this limits the natural LAG to only one previous observation. This compares unfavorably with the LAG function that has many more previous observations to draw from. However, the LAG is one of the more resource intensive functions which is why this trick is food for thought. A more elaborate paper may be published on this topic at a future date.

CONCLUSION

The RETAIN statement has been examined in great detail. Some of the lesser know details have been brought to light. In some cases where the RETAIN is clumsy an alternate coding schema was proposed. Its relationship with the LAG function was also examined. In conclusion the RETAIN statement is powerful but some hand coded techniques can be more powerful under certain conditions.

REFERENCES

CONTACT INFORMATION

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

Author Name: Venkatesan Chakravarthy
Company: Bristol Myers Squibb Co.
Address: 5 Research Pkwy, 2FW-323 Wallingford, CT 06492
Work Phone: (203) 677-3584
Fax:
Email: Venkatesan.Chakravarthy@bms.com
Web:

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.