The RETAIN Statement: One Window into the SAS® Data Step

Paul Gorrell, Westat, Rockville, MD

ABSTRACT

The behavior (or, output) of any complex system is a result of the interaction of its various subsystems. This is certainly true of SAS, and an understanding of what goes on 'behind the scenes' is an important step in mastering the use of SAS statements, functions and procedures. In this paper I will focus on the RETAIN statement, but with an eye toward certain general properties of the SAS DATA step. Looking at the DATA step from the perspective of a particular statement allows for an immediate, and concrete, grasp of what otherwise might feel like a rather abstract part of the SAS System.

Even at its best, documentation cannot delineate the full range of interaction effects a particular statement or function will exhibit as part of a DATA step or program. Understanding the general properties of the SAS System as they relate to particular parts is a prerequisite for creative and productive SAS programming, as well as a real timesaver when it comes to debugging.

INTRODUCTION

The RETAIN statement allows for comparisons between observations in a SAS data step. It is also commonly used to determine column (variable) position in a SAS data set, and to assign initial values to variables. In this paper I will discuss the RETAIN statement in detail (but, of course, not exhaustively). The first section of the paper (The SAS DATA Step) discusses general properties of the SAS system which will be important for understanding the specifics of the RETAIN statement. The second section (The RETAIN Statement: Basic Syntax) discusses the nuts and bolts of the syntactically-valid uses of the RETAIN statement. The third section (Comparing Values Across Observations) illustrates one of the more common uses of RETAIN, and briefly discusses a PROC SQL alternative. The fourth section of the paper (RETAIN: Some Efficiency Considerations) shows how the RETAIN statement can be used to decrease the processing costs of assignment statements in the DATA step. The fifth section (Determining Column Order with RETAIN) looks at how RETAIN is used to determine variable position in output SAS data sets.

THE SAS DATA STEP

The SAS DATA step has the following general characteristics (a *step* is a subpart of a program delimited by DATA or PROC and a step boundary, e.g. *RUN*).

(1) an initial *compilation* phase (performed once)

(2) a (potentially) looping *execution* phase

During compilation, (i) the syntax of all the statements in the DATA step is checked, (ii) the program data vector [PDV] is built (and an input buffer, if raw data is being read), and (iii) the descriptor portion of the SAS data set is created. Conditionals and loops aside, during execution each statement is executed once *for each observation* of the data set.

Now, let's look at compilation and execution in a bit more detail. The PDV is an area of memory where the new data set is assembled (see Whitlock 1998 for an informative discussion of the PDV and the SAS DATA step). During compilation, when a SET statement is read, the descriptor portion of the SAS data set(s) is read and each variable from the input data set(s) is given a PDV location. In addition, the automatic variables _ERROR_ and _N_ are initialized and added to the PDV. Other information is gleaned as well, e.g. the NOBS= option creates and names a temporary variable whose value is the number of observations in the input data sets (subject to certain restrictions, see the difference between NOBS and NLOBS in SAS Online Documentation).

All variables from input data sets are initially assigned missing values. Consider the DATA step in (3), which reads in the data set BEFORE1 (which has 2 variables and 3 observations). Note that the LOG OUTPUT given here is limited to the output of the PUT statement.

(3)
```
BEFORE1
V1    V2
 1     2
 3     4
 5     6
```

(4)
```
data AFTER1;
        put _ALL_;
        set BEFORE1 nobs=nobs;
run;
```

(5)
```
LOG OUTPUT:
NOBS=3   V1=.  V2=.  _ERROR_=0  _N_=1
NOBS=3   V1=1  V2=2  _ERROR_=0  _N_=2
NOBS=3   V1=3  V2=4  _ERROR_=0  _N_=3
NOBS=3   V1=5  V2=6  _ERROR_=0  _N_=4
```

Notice that understanding the distinction between the compilation and execution phases (and realizing that information is gathered during compilation) resolves what appears to be an ordering paradox in the timing of when NOBS gets a value of '3'. That is, the PUT statement initially executes before the SET statement, but (as the first line of the LOG OUTPUT shows) NOBS already has a value of '3'. Notice also that the PUT _ALL_ statement respects the order of variables in the PDV, which, in turn, reflects the order of variables in the input data set. We will return to this aspect of PDV construction when we consider the use of the RETAIN statement in determining column order in a SAS data set.

Now let's add an assignment statement to the DATA step in (4) and see how values of the new variable V3 change as each observation is read and the DATA step loops. Note the addition of a second PUT statement (a prefix of 'A' or 'B' will distinguish the output of the two PUT statements). In (6) I've removed the NOBS= option from the SET statement, as well as the LOG report of _ERROR_, as they're irrelevant here.

(6)
```
data AFTER1;
        put 'A: ' _ALL_;
        set BEFORE1;
        v3=v1+v2;
        put 'B: ' _ALL_;
run;
```

(7) LOG OUTPUT:

```
A: V1=.  V2=.  V3=.   _N_=1
B: V1=1  V2=2  V3=3   _N_=1
A: V1=1  V2=2  V3=.   _N_=2
B: V1=3  V2=4  V3=7   _N_=2
A: V1=3  V2=4  V3=.   _N_=3
B: V1=5  V2=6  V3=11  _N_=3
A: V1=5  V2=6  V3=.   _N_=4
```

As in (5) we see that the values of V1 and V2 are initially set to missing. The value of V3 is initially missing as well. Then the first observation from BEFORE1 is read and the values of V1 and V2 are read into the PDV, along with the value of V3 (product of the assignment statement). But the top of the loop for the second iteration of the DATA step (as shown by the second A line) reveals a difference between variables V1 and V2 (from the input data set listed on the SET statement) and V3 (from the assignment statement). V3 is now set to missing, while V1 and V2 have retained their previously assigned value.

This is because values of variables read in via a SET, MERGE or UPDATE statement are automatically retained, but variables whose values are initially determined by an assignment statement are set to missing at the top of the loop. The second B line shows the value of all 3 variables after the second row of BEFORE1 is read in. The DATA step repeats until the last observation of BEFORE1 is read, with the assignment statement executing for each observation (i.e. if there were 1 million observations, then the assignment statement would apply 1 million times).

The next DATA step illustrates this implicit retention when a variable is read in via a SET statement. Here there are two input data sets. Data set BEFORE1 is unchanged. Data set BEFORE2 has 1 variable (V3) and 1 observation. Its SET statement is within a conditional restricting its application to when _N_ = 1;

(8)
```
data AFTER2;
        put 'A: ' _ALL_;
        if _N_ = 1
                then set BEFORE2;
        set BEFORE1;
        put 'B: ' _ALL_;
run;
```

(9) BEFORE2

```
   V3
    7
```

(10) LOG OUTPUT:

```
A: V3=.  V1=.  V2=.   _N_=1
B: V3=7  V1=1  V2=2   _N_=1
A: V3=7  V1=1  V2=2   _N_=2
B: V3=7  V1=3  V2=4   _N_=2
A: V3=7  V1=3  V2=4   _N_=3
B: V3=7  V1=5  V2=6   _N_=3
A: V3=7  V1=5  V2=6   _N_=4
```

Once the first observation is read, there are no more missing values for any of the variables. This is because all the variables are read in via a SET statement (with its implicit retention). It is important to note that this does not mean *impervious to change or subsequent alteration*, but simply that when a value of a variable is retained, it is not set to missing when the DATA step resets in preparation for the next row of data to be read in. The value of V3 in the first row of AFTER2 persists through the subsequent rows only because there is no operation in the execution of the DATA step which would alter it.

Notice also the order of the variables in the PUT _ALL_ output. V3 is listed first. This is because it was read in by the first SET statement. Input variables are accorded positions in the PDV (and consequently in the output data sets) based on when they are encountered during compilation (see the discussion of *PDV-relevant* below). We can easily illustrate this by reversing the order of the two SET statements in the DATA step in (8). Consider the revised DATA step in (11) and the output in (12).

(11)
```
data AFTER3;
        put 'A: ' _ALL_;
        set BEFORE1;
        if _N_ = 1
                then set BEFORE2;
        put 'B: ' _ALL_;
run;
```

(12) LOG OUTPUT:

```
A: V1=.  V2=.   V3=.  _N_=1
B: V1=1  V2=2   V3=7  _N_=1
A: V1=1  V2=2   V3=7  _N_=2
B: V1=3  V2=4   V3=7  _N_=2
A: V1=3  V2=4   V3=7  _N_=3
B: V1=5  V2=6   V3=7  _N_=3
A: V1=5  V2=6   V3=7  _N_=4
```

As we discuss the RETAIN statement in the next and subsequent sections, the following properties of the DATA step will be important to bear in mind:

(13) a. The PDV is constructed during compilation, with input variables positioned from left to right in the order they are encountered in PDV-relevant statements.

b. Assignment statements are potentially executed for each observation.

c. The values of variables from assignment statements are set to missing at the beginning of each iteration of the DATA step loop.

THE RETAIN STATEMENT: BASIC SYNTAX

The core operation of the RETAIN statement is to prevent a variable's value from being set to missing from one iteration of the DATA step to the next. Why would you want to do this? One common reason is that you need to make some form of comparison between *rows* in the same way that you often do between columns. A comparison such as *if x > y* is a within-row comparison, and easy to do given that the DATA step is executing row by row. But row by row execution makes between-row comparisons less straightforward. That's where the RETAIN statement is most useful. Here's the basic syntax:

(14) retain [variable name/list | array name] [initial_value];

We will consider variations on this theme below, but for now let's minimally alter the DATA step in (6), as in (15), which illustrates the basic use of RETAIN. Here, in the absence of an overt specification, the value of V3 is initialized by the RETAIN statement as missing.

.

(15)
```
data AFTER1;
        put 'A: ' _ALL_;
        set BEFORE1;
        retain v3;
        v3=v1+v2;
        put 'B: ' _ALL_;
run;
```

(16) LOG OUTPUT:

```
A: V1=.  V2=.  V3=.  _N_=1
B: V1=1  V2=2  V3=3  _N_=1
A: V1=1  V2=2  V3=3  _N_=2
B: V1=3  V2=4  V3=7  _N_=2
A: V1=3  V2=4  V3=7  _N_=3
B: V1=5  V2=6  V3=11 _N_=3
A: V1=5  V2=6  V3=11 _N_=4
```

Notice that this output resembles that in (12). The effect of the RETAIN statement is similar to that of the SET statement: values of variables are not set to missing at the top of the DATA step loop.

The statements in (17) comprise a partial list of various syntactically-valid uses of RETAIN.

(17)
```
a. retain var1;
b. retain var1 0;
c. retain var1-var3 0;
d. retain var1 0 var2 1 var3 2;
e. retain var1-var3  ( 0 );
f. retain var1-var3 ( 0 1 );
g. retain var1-var3  ( 0 1 2 );
h. retain var1-var3  ( 0, 1, 2 );
i. retain;
j. retain _ALL_; [or _CHAR_, _NUMERIC_ ]
```

In (17b) the statement indicates that VAR1 should be (re)set with a value of '0' at the top of the DATA step loop. Similarly for the variable list in (17c), VAR1, VAR2, and VAR3 are all '0' before the next row of data is read. The statement in (17d) shows one way to assign distinct initial values to different variables. In general the lack of parentheses around the initial value indicates that that value should be assigned to all variables to its left (until another initial value is encountered).

Parentheses are used for one-to-one assignment of an initial value to a variable. For example, in (17e) the initial value of '0' is assigned only to the leftmost variable, i.e. VAR1. Following this logic, the value of '0' in (17f) is assigned to VAR1 and '1' is assigned to VAR2. In (17g), the values '0', '1', and '2' are assigned to VAR1, VAR2, and VAR3 respectively. The use of commas in (17h) is optional. Commas do not carry any meaning; they are simply an alternative delimiter for the initial-value list.

The syntax in (17i) shows that neither a variable nor an initial value is required. It is interpreted as an instruction that all variables be retained (and if you think that makes it equivalent to *retain _ALL_* then I've got a NODUP option I'd like to sell you). As the documentation outlines, the difference between (17i) and (17j) is that (17i) causes all variables to be retained, whereas (17j) affects only those variables defined before the RETAIN statement (similarly for _CHAR_ and _NUMERIC_). It's one of those 'minor' differences that can cause debugging headaches.

Variables which require an explicit RETAIN statement are those appearing on an INPUT statement and those initialized in an assignment statement. Generally, if you are reading in a variable from a SAS data set, it will be retained. Here's a list which shows when variables are automatically (implicitly) retained, i.e. retained in the absence of a RETAIN statement. With the exception of _N_ and _ERROR_, all of these variables may be assigned initial values in a RETAIN statement.

(18)
```
a.  variables read in with a SET, MERGE,
    or UPDATE statement.
b.  a variable whose value is assigned
    in a sum statement.
c.  the automatic variables _N_,
    _ERROR_, _I_, _CMD_, and _MSG_.
```

```
d.  variables created by the END= and
    IN= options.
e.  variables created by FILE and INFILE
    options.
f.  _TEMPORARY_ array elements, or those
    given initial values in an ARRAY
    statement.
```

The values of array variables are automatically retained if you have assigned initial values to them in the ARRAY statement. There is no need for a RETAIN statement in this case.

COMPARING VALUES ACROSS OBSERVATIONS

Suppose you had the following data set BEFORE3, and, as part of your DATA step, you needed to determine the maximum and minimum values of V1.

(19)

```
BEFORE3
V1 V2 V3
 3  2  1
 4  6  5
 2  1  3
 6  5  4
 1  3  2
 5  4  6
```

Here you could use the RETAIN statement as in (20).

(20)

```
data AFTER3;
       set BEFORE3 end=itsover;
       retain low_v1 10 high_v1;
       if v1 > high_v1
             then high_v1=v1;
       if v1 < low_v1
             then low_v1=v1;
       if itsover
             then put high_v1= low_v1=;
run;
```

(21)

```
LOG OUTPUT

HIGH_V1=6 LOW_V1=1
```

The retained variables HIGH_V1 and LOW_V1 are successively compared to the values of V1 in the current row. Note that the initial value of LOW_V1 is set sufficiently high that it will be altered by the data. If the default missing value is used, that will be the final value which is output. If the value of V1 is higher (or, lower) than HIGH_V1 (or, LOW_V1) then the value of the retained variable is updated. But of course there is another way to do this in SAS (SAS almost always offers more than one way to accomplish a particular programming goal.). Intuitively, you could sort the data set (ascending or descending) and output the last observation of the resulting data set (or subgroup of interest). This is fine if you are working with small data sets, but an extra sort of even a modestly-large data set is to be avoided if possible. Further, if you need more than either the highest or lowest value of a particular variable (as in (20)), you would need additional sorts.

In addition to statistical PROCs (e.g. SUMMARY), a better alternative might be the MAX and MIN functions of PROC SQL.

(22)

```
proc sql;
       create table AFTER3 as
       select max(v1) as high_v1,
              min(v1) as low_v1
       from BEFORE3;
quit;
```

Whether or not PROC SQL can serve as an alternative may depend on what you need to accomplish with the DATA step (see Gao 1999 for one interesting comparison). For example, suppose that your data set contains missing values for V1, as in (23).

(23)

```
BEFORE3  (final row added)
V1 V2 V3
 3  2  1
 4  6  5
 2  1  3
 6  5  4
 1  3  2
 5  4  6
 .  8  9
```

In this case, the output of DATA step (20) would be (24a), but the output of PROC SQL (22) would be (24b).

(24)

```
a. HIGH_V1=6  LOW_ID=.
b. HIGH_V1=6  LOW_ID=1
```

The reason for this difference is that aggregate functions such as MAX and MIN do not consider missing values. This difference may or may not be relevant to your needs. The point is simply that the more you know about the specific properties of the functions or statements you are using, and how they interact with general properties of the SAS DATA step, the better off you are in writing programs that accomplish all (and only!) what you want to accomplish. In the next section we consider RETAIN with respect to efficiency considerations.

RETAIN: SOME EFFICIENCY CONSIDERATIONS

An important fact about the RETAIN statement from an efficiency perspective is that it is NOT an instruction for SAS to do something, rather it is an instruction for SAS to NOT do something which it otherwise would. That is, SAS normally resets certain variables to missing at the top of the DATA step. This is an operation with some minimal processing cost, but it is a cost that is paid observation by observation. If you have a data set with a few million observations, then you are probably looking for ways to keep processing costs to a minimum. The RETAIN statement can play a significant role in this.

Consider the three DATA steps in (25).

(25) a.
```
data TWO;
       set ONE;
       datevar="&SYSDATE";
run;
```

b.
```
data TWO;
       set ONE;
       retain datevar;
       if _N_=1
       then datevar="&SYSDATE";
run;
```

c.
```
data TWO;
       set ONE;
       retain datevar "&SYSDATE";
run;
```

All three of these DATA steps accomplish the same thing: each observation of the data set TWO has the SYSDATE value for DATEVAR (a client once actually requested output data sets with this property). Virgile (1998) reports that, although the differences are small, there is a "consistently measurable" cost for the intuitively simple assignment statement (as in (25a)) when compared to the use of a RETAIN statement with an initial-value specification, as in (25c)). Note that Virgile (1998) compared DATA steps with 5 assignment statements and an input data set with 100,000 observations. The differences may appear too minor to bother with, but with the increasing size of data sets, interacting costs can quickly build up.

Why would (25a) use more processing resources than (25c)? Recall that an assignment statement is executed for each observation. Further, the variables defined in assignment statements are reset to missing at the top of each iteration of the DATA step. The RETAIN statement prevents this from occurring (remember, RETAIN is an instruction NOT to do something). In (25c) the variable DATEVAR is given a value once, and this value is retained for every subsequent observation. In (25b) two statements are used to accomplish the same goal. Although I have never found a measurable processing difference between (25b) and (25c), it does save a few keystrokes to use (25c).

DETERMINING COLUMN ORDER WITH RETAIN

For processes internal to SAS, column order doesn't play a significant role (e.g. I've never wished I could sort *by column*). It is usually when it comes to output that appearances matter, and PUT and VAR statements are fairly handy for that. But SAS often interfaces with applications which do care about column order. In these circumstances, it is very useful to know how to (re)arrange the order of columns in a SAS data set.

In the discussion of the PDV, we noted that input variables are positioned from left to right as they are encountered in the DATA step. This property of SAS processing can be used in conjunction with the fact that the RETAIN statement (as a non-executable statement) can appear anywhere in the DATA step. First we should note that the phrase "can appear anywhere in the DATA step" does not mean that the RETAIN statement's position relative to other statements is meaningless. It was noted above that *RETAIN _ALL_* affects only those variables defined before this statement.

In discussing the difference in column (or, variable) order resulting from the DATA steps in (8) and (11), it was the relative position of the two SET statements which determined column order. That is, the variables from the data set referenced in the first SET statement were positioned to the left of variables from the data set named on the second SET statement. The generalization being that input variables are positioned in the order they are encountered in PDV-relevant statements. Given this, we can modify the output of the DATA step in (15) by changing the relative positions of the SET and RETAIN statements, as in (26). Compare the output in (27) with (16).

(26)
```
data AFTER1;
       put 'A: ' _ALL_;
       retain v3;
       set BEFORE1;
       v3=v1+v2;
       put 'B: ' _ALL_;
run;
```

(27) LOG OUTPUT:

```
A: V3=.   V1=.  V2=.  _N_=1
B: V3=3   V1=1  V2=2  _N_=1
A: V3=3   V1=1  V2=2  _N_=2
B: V3=7   V1=3  V2=4  _N_=2
A: V3=7   V1=3  V2=4  _N_=3
B: V3=11  V1=5  V2=6  _N_=3
A: V3=11  V1=5  V2=6  _N_=4
```

In (26) the RETAIN statement occurs before the SET statement, and consequently, the variable named there (V3) is positioned first (leftmost) in the PDV and the output data set.

Care must be taken with the generalization that order of appearance in the DATA step determines PDV order. For example, referring to a variable as part of the KEEP= option in the DATA statement, or as an argument of a KEEP statement, does not affect PDV order. This is because neither use of KEEP affects PDV behavior. This can be shown by replacing the RETAIN statement in (26) with a KEEP statement. Compare DATA step (28), and LOG OUTPUT (29), with (26) and (27).

(28)
```
data AFTER1;
       put 'A: ' _ALL_;
       keep v3 v2;
       set BEFORE1;
       v3=v1+v2;
       put 'B: ' _ALL_;
run;
```

(29) LOG OUTPUT:

```
A: V1=.  V2=.  V3=.   _N_=1
B: V1=1  V2=2  V3=3   _N_=1
A: V1=1  V2=2  V3=.   _N_=2
B: V1=3  V2=4  V3=7   _N_=2
```

```
A: V1=3  V2=4  V3=.   _N_=3
B: V1=5  V2=6  V3=11  _N_=3
A: V1=5  V2=6  V3=.   _N_=4
```

Although the output data set will only contain the variables V3 and V2, it is clear that the KEEP statement did not affect the PDV. That is, the variable V1 is present for each iteration (despite its absence from the KEEP statement), and the order of variables is exactly what it would have been if no KEEP statement were present. Again, the more you know about the interacting properties of the SAS System, the less often you'll be surprised at its output (Now you know why I was careful to refer to *PDV-relevant* statements in (13a)).

One final note on column order: With the RETAIN statement, you can even 'pick and choose' from the input data set to rearrange column order, as in (30), which produces the output in (31). Here the RETAIN statement refers to one variable from the assignment statement and one from the SET statement.

(30)
```
data AFTER1;
       put 'A: ' _ALL_;
       retain v3 v2;
       set BEFORE1;
       v3=v1+v2;
       put 'B: ' _ALL_;
run;
```

(31) LOG OUTPUT:

```
A: V3=.   V2=.  V1=.  _N_=1
B: V3=3   V2=2  V1=1  _N_=1
A: V3=3   V2=2  V1=1  _N_=2
B: V3=7   V2=4  V1=3  _N_=2
A: V3=7   V2=4  V1=3  _N_=3
B: V3=11  V2=6  V1=5  _N_=3
A: V3=11  V2=6  V1=5  _N_=4
```

CONCLUSION

I've discussed four important functions of the RETAIN statement.

(32)
```
a. Comparisons between observations.
b. Assigning initial values to variables.
c. Efficient use of assignment
   statements.
d. Determination of column (variable)
   order.
```

Although in many cases PROC SQL can serve as a viable (and more readily coded) alternative to the use of RETAIN for certain between-observation comparisons, the RETAIN statement is still commonly used for this purpose, and remains a powerful DATA step tool. With respect to the efficient use of assignment statements, whenever a constant must be repeatedly assigned to a variable, the RETAIN statement is more efficient than the simple assignment statement (which must be executed for each eligible observation). Finally, the RETAIN statement (suitably positioned) is a straightforward way to arrange the order of columns in a SAS data set to meet external requirements.

Y2K AFTERWARD

In order to insure that this paper is Y2K compatible, I would urge you to replace &SYSDATE in (25) with a 4-digit year specification, e.g. &THE_DATE as defined in (33).

(33)
```
%global THE_DATE;

data _NULL_;
       call symput ("THE_DATE",
       put(input("&SYSDATE",date11.),
                     mmddyy10.));
run;
```

You can use &SYSDATE9 if you have Version 7.

REFERENCES

Gao, D. (1999) "Efficiency Techniques: SQL vs. Retain Variables," *Proceedings of the Twenty-Fourth Annual SAS® Users Group International Conference*, pp. 580-581.

Virgile, R. (1998) *Efficiency: Improving the Performance of Your SAS® Applications*. Cary, NC: SAS Institute Inc.

Whitlock, M. (1998) "The Program Data Vector as an Aid to DATA Step Reasoning," *Proceedings of the Sixth Annual Conference of the Southeast SAS®Users Group*, pp. 229-238.

ACKNOWLEDGMENTS

I would like to thank Marianne Whitlock and Mike Rhoads for comments on a draft of this paper.

CONTACT INFORMATION

Paul Gorrell
Westat, Inc.
1650 Research Blvd.
Rockville, MD 20850
email: gorrelp1@westat.com