PharmaSUG China 2021 - Paper CC-081

The Hidden Influence of Program Structure in the SAS DATA Step

Xiangyu Song, Laekna Therapeutics

ABSTRACT

The SAS DATA Step is one of the most favored processes for creating, calculating, and manipulating data in SAS. A well-structured code is of great importance for robust DATA Step programs. In pursuit of an elegant code, one needs to comprehend some special “thinking patterns” behind the SAS DATA Step. In this paper, examples are given to illustrate how small changes in the program structure can lead to different programming results by changing the position of a specific piece of code to different lines of the same DATA step. By comparing these pairs of examples, we will walk through some logics hidden behind the SAS DATA Step. Combined with discussions about how the Program Data Vector (PDV) and the Implied Loop of the DATA step (ILDS)[1] works on a case-by-case basis, these examples will be beneficial for beginner and intermediate SAS users to understand those hidden thinking patterns of the SAS DATA Step and how they can exploit, manipulate, and trick it.

INTRODUCTION

The design of the SAS DATA Step can be traced trace back to 1960–1970, when SAS was first created. The basic concepts in the construction of the SAS DATA step are to provide a handy tool for manipulating data with great flexibility. Therefore, the SAS DATA Step is smartly designed to make the data transformation process as simple as possible.

Unlike other programming languages, much of the tedious work has been set into defaults in the SAS DATA Step. These defaults frame many of the SAS DATA Step specialized thinking patterns. The SAS DATA step is processed sequentially via the compilation and execution phases. A typical SAS DATA Step process is shown in Figure 1.

Figure 1. SAS DATA Step Workflow[2]

Every DATA step starts with the compilation phase. The statements that make up the DATA step are compiled, and the syntax is checked. If the syntax is correct, then the statements are executed. In its

simplest form, the DATA step is a loop with an automatic output and return action during the execution phase.

It is essential for programmers to be aware of the different logics and actions of the SAS DATA step in the compilation and execution phases. This paper will focus on revealing those hidden thinking patterns of the SAS Data Step in this action flow by comparing pairs of examples. Within each pair of example code, Code A and Code B are nearly identical in the content of the statements. The only difference between them is the structure of the program and the order of the statements. By walking through these examples, we will be able to test out the thinking logics of SAS when processing the DATA Step.

COMPILATION PHASE

In this phase, each statement is scanned for syntax errors. Afterwards, SAS uses the PDV, a logical area of memory on computers, to build an ideal data set. The PDV holds information regarding inputting and the input formatting of desired variables. In the compilation phase, the DATA step code is scanned from top to bottom as a single vertical pass through the PDV, and the information received from this scan is used to create the descriptor portion of an ideal SAS data set.

A variable and its characteristics are defined by the PDV using the first occurrence of the variable. Thus, the physical placement of the line where a variable first occurs determines how PDV will build the ideal data set.

Although invisible to programmers, this workflow influences the output data set in many respects.

EXAMPLE 1: VARIABLE ORDER AND ATTRIBUTES

Code A and code B in this example have identical statements. The only difference between them is the location of the length statement, as follows:

A:

```
data A;
x=123;
y=”abc”;
z=”a3b2c1”;
length x 8 y $8 z $12;
run;
```

B:

```
data B;
length z $12 x 8 y $8;
x=123;
y=”abc”;
z=”a3b2c1”;
run;
```

However, the output data sets are different, as shown in Figure 2.

Data A

Obs	x	y	z
1	123	abc	a3b2c1

Data B

Obs	z	x	y
1	a3b2c1	123	abc

Figure 2. Example 1 Output Data

In Data B, z is the first column of data, whereas in Data A, the column order is x, y, and z.

The explanation for this is that while the values of x, y, and z are assigned in the same order in A and B, the length statement `length z $12 y $10 x 8` is placed first in B. This occurrence of z comes first in the vertical scan of the PDV, so z is the first variable the compiler writes into the ideal data set. However, in A, the value assignment statements `x=123; y="abc";z="a3b2c1";` occur first, so x, y, and z are created in that order.

Apart from the variable order, the type and length of each variable is identified by its first occurrence. The length of y and z are different in the two data sets, as shown in Figure 3.

Data A

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	x	Num	8
2	y	Char	3
3	z	Char	6

Data B

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	x	Num	8
3	y	Char	8
1	z	Char	12

Figure 3. Example 1 Variable Attributes

In Code A, y and z are created by `y="abc";z="a3b2c1"` statements. The PDV encounters them first during the scan, and because no length statement is specified at that time, the PDV assigns a default length to y and z, which is the length of "abc" and "a3b2c1". The later length statement is omitted until executing this code. A warning message will be written to the log during the execution phase:

```
WARNING: Length of character variable z has already been set.
```

A good practice is to set length at the top of your program in case of any truncation of character variables. This logic also applies when creating new data from a parent SAS data set using a SET statement.

A:

```
data A_;
SET B;
RETAIN x y z;
run;
```

B:

```
data B_;
RETAIN x y z;
SET B;
run;
```

Data A_

Obs	z	x	y
1	a3b2c1	123	abc

Data B_

Obs	z	x	y
1	123	abc	a3b2c1

Figure 4. Example 1 Output 2

Only Data B's variable order was changed by the RETAIN statement. This is because, in Code A , the SET statement is placed before the RETAIN statement; thus, the first occurrence of x, y, and z comes from Data B. Therefore, the variable order in the new data set A__ is inherited from the parent data set B.

EXAMPLE 2: DECLARATIVE STATEMENT AND DATA STEP OPTIONS

In the SAS DATA step, some statements are compiled time only. They provide declarative information to the PDV for it to know how to build the data in logic. If any error or inconsistency is found in these declarative statements, the data step will not start execution. Declarative statements supply information to SAS and take effect when the system compiles program statements.

Although declarative statements can be placed in any order in the DATA step, they are always processed before the execution statement.

Take KEEP WHERE DROP RENAME as examples. KEEP DROP WHERE RENAME is a special set of declarative statements that share the same key words with DATA STEP options (WHERE=, KEEP=, DROP=, RENAME=). Although they look similar and have overlapping functions in many cases, programmers need to be aware that they are inherently different. The difference between the declarative statement and DATA step options can be demonstrated in the next example:

A:

```
data car_a car_b ;
   set sashelp.cars(keep=Cylinders);
   output car_a;
   drop Cylinders;
   output car_b;
   run;
```

B:

```
data car_a car_b(drop=Cylinders);
  set sashelp.cars;
  keep Cylinders;
  output car_a;
  output car_b;
  run;
```

At first glance, the output of Code A seems to be a car_a data set with one column, “Cylinders,” and an empty car_b data set.

However, the actual running results of Code A are shown in log :

```
/*------A----------------------------------------*/
data car_a car_b ;
set sashelp.cars(keep=Cylinders);
output car_a;
drop Cylinders;
output car_b;
run;

NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CAR_A has 428 observations and 0 variables.
```

```
NOTE: The data set WORK.CAR_B has 428 observations and 0 variables.
NOTE: DATA statement used (Total process time):
real time           0.00 seconds
cpu time            0.00 seconds      cpu time            0.00 seconds
```

Log 1.

Both the data set car_a and the data set car_b are empty.

It is only when DROP is moved into parentheses as a data set option (Drop=Cylinders) in code B that the cylinder column will be outputted to car_a. This is because in code A, `drop Cylinders` is a declarative statement that, regardless of its line placement, affects all of the output data sets. In contrast, in code B, (drop=Cylinders) is a data step option that only affects data sets it is attached to.

EXAMPLE 3: THE PLACE OF RETAIN

The RETAIN statement is also a declarative statement, but it is slightly different from WHERE DROP KEEP RENAME, which information is interpreted by the PDV as one of the characteristics of variables. The PDV divides variables into two groups according to the RETAIN statement.

One is the retained group, where the PDV keeps the last value it had when a new iteration starts. The other is the not retained group, where the PDV sets them as missing when a new iteration starts.

The not retained group includes:

- The values of variables coded in an INPUT statement
- User-defined variables (e.g., x=, y=, z=)

Variables in the retained group include:

- SAS special automatic variables
- Variables coded in a RETAIN statement
- Variables read with a SET, MERGE, or UPDATE statement
- Accumulator variables in a SUM statement.

This grouping process happens at the same time as the PDV creates the variables in its logic. Below is an example using the A and B data sets we created in Example 1.

Data A

Obs	x	y	z
1	123	abc	a3b2c1

Data B

Obs	z	x	y
1	a3b2c1	123	abc

A:

```
Data A__;
   retain _ALL_;
   set A(in=a) B;
   if a then do;
   x2=123;
   end;
   run;
```

B:

```
Data B__;
   set A(in=a) B;
```

```
if a then do;
x2=123;
end;
retain _ALL_;
run;
```

Figure 5 presents the results of A and B:

DATA A__

Obs	x	y	z	x2
1	123	abc	a3b2c1	123
2	123	abc	a3b2c1	.

DATA B__

Obs	x	y	z	x2
1	123	abc	a3b2c1	123
2	123	abc	a3b2c1	123

Figure 5. Example 3 Outputs

The explanation for these results is that in Code A, the statement RETAIN _ALL_ is placed before the first occurrence of variable x2. Thus, the PDV does not know there will be a variable x2 at the time it grouped all the variables into the retained group. As a result, because it first appeared in the user-defined statement x2=123, x2 was marked as a not retained variable by the PDV. As a result, x2 is set to missing before SAS reads the second line of data.

In Code B, the first occurrence of x2 is before the RETAIN _ALL_ statement, so it is sorted into the retained group. As a result, the value of x2 is kept from the first row when SAS reads the second line from the B data set.

EXECUTION PHASE

The execution phase only begins after the compilation phase ends.

In the execution phase, the DATA step works like a loop, repetitively executing statements. Each loop is called an iteration. This looping process is integrated into SAS defaults and is implicit to programmers, so it is called the implicit loop of DATA step (ILDS).

If we marked out the statements of SAS defaults in this looping process, the SAS DATA[4] step would appear as shown in Figure 6:

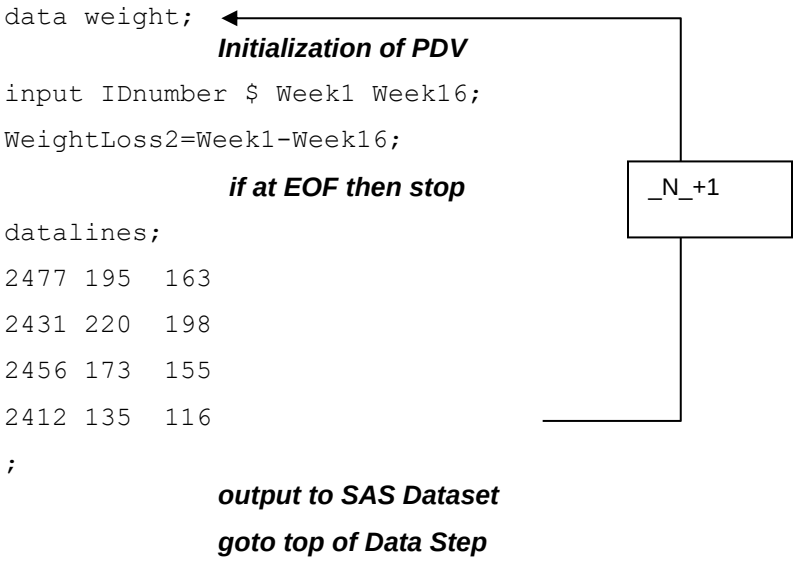

Figure 6. The SAS DATA Step with Defaults Marked Out

The automatic variable _N_ can be used to identify the number of the ILDS.

It is noteworthy that, although it is a common practice to use _N_ to assign the row number of a data set, the variable _N_ stands for the time of the Implied Loop of the DATA Step (ILDS). Thus, one should be very careful when try to build up logic for calculation based on _N_ .

EXAMPLE 4: WHERE ILDS STOPS

With a basic understanding of the form of the ILDS, the next question is how the end loop condition is determined by the SAS DATA step without any end loop condition statement. As shown in Figure 1, the switch of the data step looping is the mark of no more input data.

An example demonstrating the default logics that SAS uses to end the loop is as follows:

A:

```
Data CAR_A;
   do i=1 to 5;
   set sashelp.cars(obs=2 KEEP=Make Model Type Origin DriveTrain) ;
   output;
   end;
   run;
```

B:

```
Data CAR_B;
   set sashelp.cars(obs=2 KEEP=Make Model Type Origin DriveTrain);
   do i=1 to 5;
   output;
   end;
   run
```

The outputs are shown in Output 1.

Car_A

Obs	i	Make	Model	Type	Origin	DriveTrain
1	1	Acura	MDX	SUV	Asia	All
2	2	Acura	RSX Type S 2dr	Sedan	Asia	Front

Car_B

Obs	Make	Model	Type	Origin	DriveTrain	i
1	Acura	MDX	SUV	Asia	All	1
2	Acura	MDX	SUV	Asia	All	2
3	Acura	MDX	SUV	Asia	All	3
4	Acura	MDX	SUV	Asia	All	4
5	Acura	MDX	SUV	Asia	All	5
6	Acura	RSX Type S 2dr	Sedan	Asia	Front	1
7	Acura	RSX Type S 2dr	Sedan	Asia	Front	2
8	Acura	RSX Type S 2dr	Sedan	Asia	Front	3
9	Acura	RSX Type S 2dr	Sedan	Asia	Front	4
10	Acura	RSX Type S 2dr	Sedan	Asia	Front	5

Example 4 Outputs 1 When SAS determine the end

In both Code A and Code B, the end of the DO loop is set as 5. But the car_A data set has only two records, and if we look at variable I, it appears that the DO loop stopped at i=2. This is because the DO loop is constructed differently in A and B.

In Code A, the DO statement is placed before the SET statement, and it is also before the hidden statement ***if at EOF then stop***. In addition, the END statement of the DO loop is before the ***goto top of Data Step.*** This means the SET statement will be rerun with the increment of i. But why are there only two rows in the output data set?

Let us walk through the thinking process of SAS when reading Code A by adding some PUT statements and marking the end condition with an END=ifend statement:

```
Data CAR_A;
put _ALL_;
   do i=1 to 5;
set sashelp.cars(obs=2 KEEP=Make Model Type Origin DriveTrain) END=ifend ;
put _ALL_;
   output;
   put _ALL_;
   end;
   run;
```

The log is presented in below:

```
① i=. ifend=0 Make=  Model=  Type=  Origin=  DriveTrain=  _ERROR_=0 _N_=1
② i=1 ifend=0 Make=  Model=  Type=  Origin=  DriveTrain=  _ERROR_=0 _N_=1
③ i=1 ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All
_ERROR_=0 _N_=1
④ i=2 ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All
_ERROR_=0 _N_=1
⑤ i=2 ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front _ERROR_=0 _N_=1
⑥ i=3 ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front _ERROR_=0 _N_=1
NOTE: There were 2 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CAR_A has 2 observations and 5 variables.
NOTE: DATA statement used (Total process time):
real time           0.00 seconds
cpu time            0.00 seconds
```

Curiously, only one ILDS loop was created. The automatic variable _N_ is always 1. If we locate each put statement from the log message, then we have the following:

```
Data CAR_A;
   Initialization of LPDV
   put _ALL_; ①
```

```
do i=1 to 5;
put _ALL_; ② ④ ⑥
set sashelp.cars(obs=2 KEEP=Make Model Type Origin DriveTrain) END=ifend ;
if at EOF then stop
put _ALL_; ③ ⑤
output;
end;
goto top of Data Step
run;
```

The anatomy of each step is:

1. The program is compiled, and the PDV initialized.
2. The DO loop starts, and i is set to 1. All other variables are missing.
3. The first observation has been read from SASHELP.CAR, and the file end detection variable (END=ifend) is FALSE because it detected the second observation from SASHELP.CAR i=1 _N_=1 ifend=0. However, the first ILDS has not finished at this time because the DO loop i is still 1 the end condition is not met.
4. The DO loop starts again after the first record is outputted The DO loop i is set to 2, and _N_ is still 1. The value reads from the last SET statement are retained.
5. The second observation is read from SASHELP.CAR, and the automatic file end detection variable (END=ifend) is TRUE this time because no next record in the input dataset. Thus, i=1 _N_=1 ifend=1. The ILDS end condition was met, and it tells the program to stop at the current looping, which is _N_=1.
6. The DO loop starts again after the second record is outputted. But this time, because endif=True, the program will not read the data set again, and the program stops at _N_=1.

In contrast, in Code B, the ***if at EOF then stop*** condition is reached after the first DO loop is completed. The SET statement is placed outside the DO loop; therefore, it will not rerun with the change of i. The file end indicator ifend changes to 1 after the first DO loop is completed (① in the log shown in Output 4). The program stops at _N_=2, and Data B has 10 rows in total (five from each loop).

```
Data CAR_B;
put _ALL_;
set sashelp.cars(obs=2 KEEP=Make Model Type Origin DriveTrain) END=ifend ;
put _ALL_;
do i=1 to 5;
put _ALL_;
output;
end;
run;
```

```
ifend=0 Make=  Model=  Type=  Origin=  DriveTrain=  i=. _ERROR_=0 _N_=1
ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=.
_ERROR_=0 _N_=1
```

```
ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=1
_ERROR_=0 _N_=1

ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=2
_ERROR_=0 _N_=1

ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=3
_ERROR_=0 _N_=1

ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=4
_ERROR_=0 _N_=1

ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=5
_ERROR_=0 _N_=1

ifend=0 Make=Acura Model=MDX Type=SUV Origin=Asia DriveTrain=All i=.
_ERROR_=0 _N_=2

① ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=. _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=1 _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=2 _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=3 _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=4 _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=5 _ERROR_=0 _N_=2

ifend=1 Make=Acura Model=RSX Type S 2dr Type=Sedan Origin=Asia
DriveTrain=Front i=. _ERROR_=0 _N_=3

NOTE: There were 2 observations read from the data set SASHELP.CARS.

NOTE: The data set WORK.CAR_B has 10 observations and 5 variables.

NOTE: DATA statement used (Total process time):

real time           0.00 seconds

cpu time            0.00 seconds
```

Output 4. <Insert appropriate title/legend>

EXAMPLE 5: WHERE TO PLACE WHERE=

As we briefly discussed in the compilation phase section, the placement of the DATA step options will only affect those data they are attached to. But with which data to place these options also matters in many cases. The placement of the WHERE= statement is an example.

Consider that we have raw data collected on blood pressure for patients by week and weekdays, as follows:

```
data SYSBP;
   input ID $ Week SYSBP Day;
   datalines;
   1  1  195  3
```

```
1  1  194  3
1  1  198  3
1  1  196  5
1  2  195 2
1  2  198 2
2  1  144  3
2  1  140  3
2  2  142  1
2  2  138  1
3  1  133  3
3  2  131  2
;
run;
```

If we want to know whether a patient's last blood pressure reading was greater than 140 on week 1, day 3, we can have Code A and Code B:

A:

```
data dataA;
   set SYSBP(where=(week=1  and sysbp>140 and day=3));
   by id week day;
   if last.day ;
   run;
```

B:

```
data dataB(where=(week=1  and sysbp>140 and  day=3));
   set SYSBP;
   by id week day;
   if last.day;
   run;
```

The differences are presented in Figure 7.

Data A

Obs	ID	Week	SYSBP	Day
1	1	1	198	3
2	2	1	144	3

Data B

Obs	ID	Week	SYSBP	Day
1	1	1	198	

Figure 7. Example 5 Outputs

Code A and Code B look the same in all respects except for the placement of the WHERE data set options. However, obviously, Code A gives the wrong output, where the first reading for patient 2 is selected.

In Code A, the `(where=(week=1  and sysbp>140 and day=3))` option goes first in every execution loop when reading the input data. Only rows that meet the criteria `week=1  and sysbp>140`

`and day=3` will be read in and executed by the order defined as `id week day`. If it is the last record in the `by` group, then it will be outputted. In B, every line of the input data set will be read in and executed by `id week day`, and if it is the last record in the `by` group, then it will be screened to see if it meets the criteria `week=1 and sysbp>140 and day=3`. The total looping number in Code B will be identical to the row number in the data set SYSBP.

However, If we map the looping number of _N_ in Code A by adding the variable row=_N_, it will appear as shown in Figure 8.

Data A

Obs	ID	Week	SYSBP	Day	row
1	1	1	198	3	3
2	2	1	144	3	4

Figure 8. Example 5 Outputs 2

It is clear that the second line of this output (week 1 day 3 record for patient 2) is neither the fourth row in the input data set nor the fourth row in the output data set. Again, while it is a common practice to use _N_ to assign the row number of a data set, he _N_ stands for the time of the Implied Loop of the DATA Step (ILDS. Here, _N_=4 means this row is in the fourth loop of Code A.

In a lot of studies in the body of literature, the use of WHERE= options in the input (SET) statement first is recommended as a favorable practice because it can increase programming efficiency and reduce processing time. But we also must have a good understanding of what the coding purpose is before doing so.

EXAMPLE 6: WHY MERGE ALWAYS WENT WRONG

As mentioned in the previous discussion, variables read with a SET, MERGE, or UPDATE statement will automatically be retained by the PDV before the next input value is found. Programmers need to code with caution when trying to integrate the MERGE step with complex logics calculations or data transforming into one SAS DATA step code, especially if there is not a crystal clear view of what the input data are.

Consider that we have two data sets we want to merge, demographic and vital signs, and we also want to recalculate weight for each patient for some reason. Figure 9 provides an example of the source data.

Demographics:

Obs	ID	NAME	SEX	WEIGHT
1	1	John	M	80
2	2	Lily	F	66
3	3	Alice	F	60

Vital Sign

Obs	ID	VIST	HR
1	1	1	80
2	1	2	100
3	2	1	89
4	2	2	67
5	3	1	66

Figure 9. Example 6 Source Data

We can run code A and achieve the output shown in Figure 10.

A:

```
data all;
   merge demog vital;
   by id;
   weight=weight*2;
   run;
```

Obs	ID	NAME	SEX	WEIGHT	VIST	HR
1	1	John	M	160	1	80
2	1	John	M	320	2	100
3	2	Lily	F	132	1	89
4	2	Lily	F	264	2	67
5	3	Alice	F	120	1	66

Figure 10. Example 6 Output Data A

In the first BY group, the PDV read the first row from demographics data that includes NAME=John, SEX=M, and WEIGHT=80, and then the first row is read from vital signs data. After this, the statement `weight=weight*2` was executed, and the first row was written into the output data set. The PDV now has NAME=John, SEX=M, and WEIGHT=160.

Next, the program starts from the top again, and _N_ is now 2. However, there are no new records from demographics, so the last value is retained: NAME=John, SEX=M, and WEIGHT=160. After the second record is read from the vital signs, the statement `weight=weight*2` was executed again, and we therefore have the value 320 in the data set. The WEIGHT value will continue to be repeatedly multiplied by 2 until the end of the BY group is reached.

If we separate this into two steps as Code B, then the result will look correct.

B:

```
data all;
   merge demog vital;
   by id;
   run;
   data all;
   set all;
   weight=weight*2;
   run
```

Obs	ID	NAME	SEX	weight	VIST	HR
1	1	John	M	160	1	80
2	1	John	M	160	2	100
3	2	Lily	F	132	1	89
4	2	Lily	F	132	2	67
5	3	Alice	F	120	1	66

Figure 11. Example 6 Output Data B

It is advisable to be very careful when adding complex logic to a DATA step that performs a merge. One should clearly understand how the PDV works and the ramifications of the automatic RETAIN. If these concepts are unclear, or one simply wishes to play it safe, the additional logic should be moved to a separate DATA step.

CONCLUSION

Knowing how to place the statement in the right place is one of the fundamental parts of efficient and successful coding. One must have a good understanding of the hidden thinking patterns of the SAS DATA step and know how to organize the code in a way that makes SAS correctly understand. It is the syntactic constituency of the SAS DATA step programming language.

REFERENCES

1. Whitlock, I. 2006. “How to Think Through the SAS DATA Step.” *Sugi 31*, paper 246-31.

2. SAS Help Center. n.d. “How the DATA Step Processes Data.” Accessed November 3, 2021. https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lepg/p0l9z26c8qrhl8n1hqs6raontpbn.htm.

3. Johnson, J. 2012. “The Use and Abuse of the Program Data Vector.” *SAS Global Forum 2012*, paper 255-2012.

4. First, S. 2009. “Understanding the SAS DATA Step and the Program Data Vector.” *SAS Global Forum 2009*, paper 136-2009..

5. Horstman, J. M. 2018. “Merge with Caution : How to Avoid Common Problems when Combining SAS Datasets.” *SAS Global Forum Proceedings*, paper 1746-2018.

CONTACT INFORMATION

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

Xiangyu Song

Songxiangyu1992@gmail.com