BASUG 2017
An Animated Guide: Learning The PDV via the SAS® Debugger™

Russ Lavery, Contractor, Bryn Mawr, PA

ABSTRACT

The program data vector can be a little bit difficult to understand because it is, normally, invisible as it does it work. It is thought that if we were able to see the program data vector itself, then understanding the PDV rules of operation would be easier and that is what this paper attempts to do. This paper attempts to merge, some of the, treatment of the PDV in Jim Johnson's *Use an abuse of the Program Data Vector* with the interactivity of the Data Step Debugger (DSD).

This paper will spend some time explaining the debugger but a more in-depth treatment can be found on the web in the paper An Animated Guide:The Data Step Debugger http://www2.sas.com/proceedings/forum2007/121-2007.pdf (there are multiple versions - look for the paper from SAS Global Forum). If a programmer wants to study the PDV, it is suggested that he read this paper and apply some of the techniques explained there.

So the benefit of reading this paper is that it connects the reader with some excellent material on PDV problems ,through papers in the reference section, and provides a new tool so that the reader can be free to discover on their own the rules set forth in the reference papers. It is hoped that no reader will finish reading this whole paper. The hope is that, after a few pages, a reader will say quote I can do that" and want to independently use the debugger to explore the PDV. BAUG is hosting this paper, and an associated file of SAS code, that can be used to explore the date step functioning. Downloading the SAS code associated with this paper can save the reader the time it takes to create examples for learning the DSD and the PDV.

INTRODUCTION

The SAS PDV is not a topic for a fainthearted author. One can take a small part of the functionality of the PDD and cover it well in ten pages but that is not covering the topic of the PDV and one should wonder if the topics can be covered in less than a book.

I often apply combinatorics to get some idea of the complexity of a new problem. I did that as I approached this topic and was terrified. Let's do a bit of combinatorics to try and get some idea of the size of the PDV as a topic. Think just in terms of three data set options drop, keep and rename. Use combinatorics to estimate how long a paper must be to just show examples of the functionality of Drop, keep and rename and to create rules to explain how they interact.

These three options can appear in multiple places in SAS code and, in each place, can appear singly, as pairs of options, or all three together as a triplet. If the options are in one place, the order in which they appear is important because different things happen when you change the order (some of which are errors).

Just the three options together, as a triplet, can be ordered six different ways and would require six different examples. To add to the complexity, each of the options can be put inside the data step in three places: 1) in the data step, 2) as a data set option on the set-merge-update data set or 3) as an option on the data set being created by the data step. There is more complexity. When we merge we can merge two or more data sets – each having these multiple ways to use the options.

Finally the options do not have to appear *together*, in one spot, in the code. The options can appear simultaneously in the data step body, the set – merge – update data set or on the data set being created by the data step. It would be very difficult to show examples of all of the possible ways in which this simple bit of SAS coding can affect the program data vector.

Accordingly; this fainthearted author will point the reader towards some excellent references and suggest that the reader use examples from those references, and the data step debugger, to ***see*** how the program data vector functions. This paper will illustrate several examples of PDV execution but does not pretend to cover the entire topic of the PVD. The contribution of this paper will be to point to existing and excellent resources that use "printed paper format" to explain PDV principles and to suggest a reader use the Data Step Debugger to independently re-discover those principles.

The SAS display manager, and other SAS interfaces, do an excellent job of capturing syntax errors. The notes in the SAS log are very helpful in identifying, and quickly fixing, syntax errors. Even newcomers can write SAS code that contains no syntax errors. If the job runs, it has no syntax errors. Syntax errors are easy to find and fix but logic errors can be very difficult to find and fix.

When your code has a logic error your code runs – and does what you instructed it to do – but what you instructed it to do is not what you wanted it to do. When this situation occurs the problem often lies with a misunderstanding of the program data vector rules.

When your code has a logic error the data step debugger can be invaluable. One should mention that since SAS executes your programs in "paragraphs" (PROCs or data steps) the debugger does not find ***all*** logical errors. If your logical error is somewhere between two "paragraphs" (perhaps, in a *previous step*, you had typed "where sex = 'F'" instead of "Where sex = 'M'"), the debugger cannot help you, much, with that problem. The d will the back ebugger is designed to help you find logic errors ***inside*** one the code for one data step.

HOW THE PROGRAM DATA VECTOR IS CREATED?

When you highlight code and hit F3, the code is sent to the input stack where it is tokenized and then sent through the word queue to the compiler.

(For a more complete description please see: An Animated Guide: The Map of the SAS® Macro Facility http://www.lexjansen.com/nesug/nesug02/bt/bt005.pdf).

When a step boundary (run; quit; PROC or Data) reaches the compiler the compiler takes control of the SAS system and performs its compile functions.

A picture of the process is in Figure 1. Some variables are valued in the compile step.

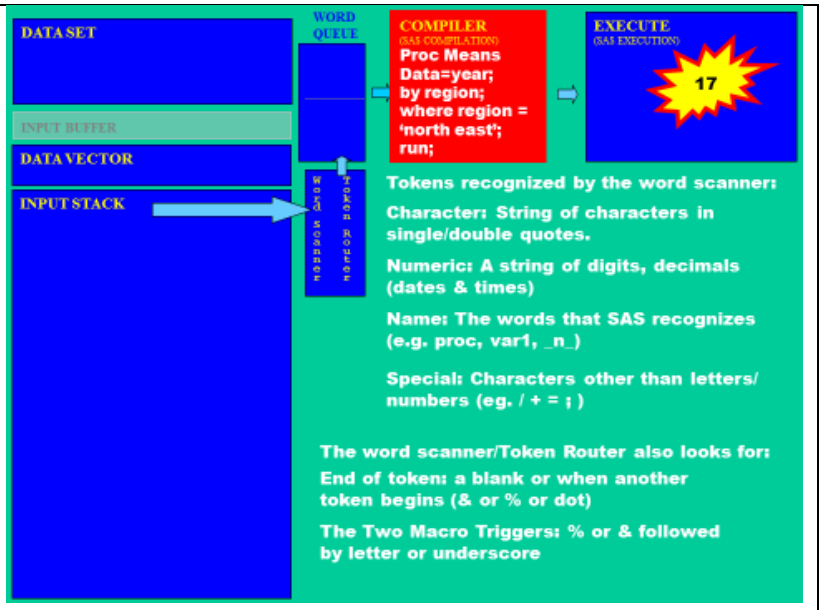

Figure 1

The compile functions are:
translates the SAS statements into machine code
reads the code from top to bottom and creates the program data vector (PDV)
the PDV includes automatic variables like: _n_, _ERROR_, End=, IN=, First. Last and POINT
spaces for variables are placed on the PDV in the order in which the compiler "sees them"
if the data step reads a SAS data set, variable information is taken from the SAS table header.
If the data step reads an external file, the compiler creates the input buffer
If the data step reads an external file, variable information is taken from the input statement
if variables are created by programming steps the compiler finds characteristics from either:
SAS defaults for that datatype
SAS statements like length

creates the descriptor, or header, section of the output file
metadata comes from different sources, and are in conflict, the compiler resolves this issue
and processes statements that are “compile time” statements
examples of these are: drop; keep; rename; retain; where; label; length; format; array; by; attrib

I think of the PDV as being a one line Excel spreadsheet. The PDV is a ***logical*** area in memory (with four physical memory locations) and all calculations happen inside the program data vector. The program data vector is unusual in that it exists in the compile step as well as in the execution phase of a SAS job.

If SAS code compiles, it is sent to the execution phase. In this phase data is read into the input buffer, if appropriate, and into the PDV. It is during the execution phase where confusion about how the PDV works causes logical errors. We will focus on the PDV during execution phase during the rest of this talk.

*RULES FOR PDV FUNCTIONING

Personally; I find that long lists of rules can be confusing. For me to understand, I typically need to see the rules written out and then to see an example of the rules being applied. That will be the method used in the rest of this paper, but in this section we will have to state some of the rules. An excellent, and long, description of the debugger rules is in Jim Johnson’s twenty-two page paper and it would be pointless to repeat all that material. Nancy Brucken explains the DOW loop in *One-step Change from Baseline Calculations and other DOW-Loop Tricks*. The contribution of this paper to the SAS community is the suggestion that these topics can be easily investigated using the data step debugger.

**The compiler reads the data step code, from top to bottom, and creates space on the program data vector for variables. The variable order (think left to right) on the program data vector is the order in which the compiler encounters the variable. The compiler can encounter a variable by the code containing a set, merge, modify, update statement – in which case the compiler will read the header for the SAS data set. The compiler can also encounter a variable because it is in a SAS statement (length, assignment, input and others).

**When the data step starts to execute for the first time the compiler initializes _N_ to 1and _ERROR_ to 0;

**Other variables are set to missing
** The compiler creates space for the variables : end, in, first, last, point and variables created by array commands
** The data step is an implied loop with statements being executed one time per loop.
** At the top of the loop variables created in the data step, and not controlled by a retained statement, are reset to missing .
** Variables read from a set – merge – modified – update statement are automatically retained.
** _N_ is incremented at the top of the data step.
** _ERROR_ is set to zero at the top of the data step.

*USE THE DEBUGGER TO EXAMINE AND LEARN THE PROGRAM DATA VECTOR

The data step debugger was one of the most requested items on the SAS software ballot from 1987 to 1991 but it is largely unknown today. The data step debugger is designed to allow people to fix logical because they can examine the program data vector ***as code executes***. Importantly; the debugger ***does not point out errors***. The debugger shows what ***is*** happening in the PDV and the programmer must compare ***what is happening*** to his/her mental model of ***what should be happening***.

SAS compile and execute modes will help a programmer debug syntax errors by “complaining” and putting notes in the log. SAS can do this because it has an idea of how syntax should be constructed and executed. However; SAS has no idea of how the data step logic should be constructed and cannot issue any helpful notes.

As was said before, using the debugger is the process of comparing ***what is happening*** (which is almost 100% of the time what SAS documentation described) with what the programmer ***wanted to have happen***. Therefore; an understanding of the program data vector is crucial in using the debugger.

The data step debugger is not a “use every day” tool. I use it every few months. It speeds up the debugging of data step sections – especially when the sections are complicated (think triple nested loops). You can watch the results of your logic, as it is being executed, by displaying values from the PDV. This makes it fantastic for learning SAS, for

learning SAS tricks, for learning the execution of the PDV and especially for learning whatever the heck Paul Dorfman is talking about in his most recent paper.

The debugger works very well on Windows machines and Unix Display Mangers. It has a relatively small set of commands to issue but, be warned, issuing individual commands makes using the debugger a bit annoying because of the large amount of typing required. To make the use of the debugger pleasant, and fast, one must learn ***combinations*** of commands – and there are, luckily, just a very few to learn. The most important to learn are : ex _all_, <Enter> <F4>, key mapping and macros. The explanation of those is left to the paper mentioned in the abstract.

USING THE DEBUGGER ON MATERIAL IN THIS TALK

It is suggested that readers of this paper copy access the SAS code hosted by BASUG and run the debugger on the over 1000 lines of code in the SAS file associated with this paper.. It is believed that this method would allow a reader to quickly duplicate all of the examples, and see all the principles, discussed in this paper. The remainder of this paper will be a collection of examples of how the program data vector works illustrated through that SAS code and screen prints from running the code.

VARIABLES CREATED BY THE COMPILER

The code to the right sets the stage for the first example. It sorts sashelp.class by sex and age and then uses SQL to create two macro variables.

To turn on the debugger one simply has to add the string "/DEBUG" to this data statement. Please see the text with the yellow background below

"/DEBUG" turns on the debugger when the section of code is run. One common mistake when learning to use the debugger is to turn on the debugger but forget to delete the code "/DEBUG" when the code is rerun the next time. So long as"/DEBUG" is in the code SAS will start the debugger every time the code is run. Delete the "/DEBUG" when you are finished using the debugger.

There are only a few debugger commands that one really needs to know. This paper will talk about the basic commands that will be used in the examples below.

A more complete treatment of the debugger can be found in the paper mentioned in the abstract.

To facilitate a quick playing/learning, all code used in this paper is in a file hosted by BASUG and the reader is encouraged to load the code from this file into their display manager and use the DSD to "play" with the examples and concepts developed in this paper.

```
Proc sort data=sashelp.class
          out=Sorted_Class;
 by sex age;run;

Proc SQL;
select avg(height), avg(weight)
  into :Avg_ht, :avg_wt
  from Sorted_Class ; quit;
238  %put Avg_ht=&Avg_ht avg_wt=&avg_wt;
     Avg_ht=62.33684 avg_wt=100.0263
```

Obs	Name	Sex	Age	Height	Weight
1	Joyce	F	11	51.3	50.5
2	Jane	F	12	59.8	84.5
3	Louise	F	12	56.3	77.0
4	Alice	F	13	56.5	84.0
5	Barbara	F	13	65.3	98.0
6	Carol	F	14	62.8	102.5
7	Judy	F	14	64.3	90.0
8	Janet	F	15	62.5	112.5
9	Mary	F	15	66.5	112.0
10	Thomas	M	11	57.5	85.0
11	James	M	12	57.3	83.0
12	John	M	12	59.0	99.5
13	Robert	M	12	64.8	128.0
14	Jeffrey	M	13	62.5	84.0
15	Alfred	M	14	69.0	112.5
16	Henry	M	14	63.5	102.5
17	Ronald	M	15	67.0	133.0
18	William	M	15	66.5	112.0
19	**PHILIP**	**M**	**16**	**72.0**	**150.0**

FIGURE 2

```
Data Example1 (drop= AverageHt AverageWt ) /debug ;
   Retain Retained_Obs_No  Ret_N_Zro_obs_no;
   array Vital(2) AverageHt AverageWt (&Avg_ht, &avg_wt);
   AboveSet="Un"||"usual";
  set sorted_Class(drop=age rename=(Name=StName)) end=eof NOBS=OBS;
  by sex;
  If mod(_n_,2)=0 and sex="F" then LagName=lag1(StName);

    if _n_=1 then Obs_No=0; Else Obs_No=Obs_No+1;
    if _n_=1 then Ret_N_Zro_obs_no = 0;
       Else Ret_N_Zro_obs_no=Ret_N_Zro_obs_no+1;
    Retained_Obs_No=Retained_Obs_No+1;
    Simple_Obs_No+1;

  ArrayMin=LBound(Vital);
  ArrayMax=HBound(Vital);
  ArrayDim=Dim(Vital);
  Ht_Diff=height-vital(1);
  Wt_Diff=weight-AverageWt;
  IF _N_=1 THEN CALL SymPutX("ObsCnt",obs);
  run;
Proc Print Data=Example1; run;
```

		Ret_N													
	Retained	Zro_obs	Above					Lag		Simple_	Array	Array	Array		
Obs	Obs_No	no	Set	StName	Sex	Height	Weight	Name	Obs_No	Obs_No	Min	Max	Dim	Ht_Diff	Wt_Diff
1	.	0	Unusual	Joyce	F	51.3	50.5		0	1	1	2	2	-11.0368	-49.5263
2	.	1	Unusual	Jane	F	59.8	84.5		.	2	1	2	2	-2.5368	-15.5263
3	.	2	Unusual	Louise	F	56.3	77.0		.	3	1	2	2	-6.0368	-23.0263
4	.	3	Unusual	Alice	F	56.5	84.0	Jane	.	4	1	2	2	-5.8368	-16.0263
5	.	4	Unusual	Barbara	F	65.3	98.0		.	5	1	2	2	2.9632	-2.0263
6	.	5	Unusual	Carol	F	62.8	102.5	Alice	.	6	1	2	2	0.4632	2.4737
7	.	6	Unusual	Judy	F	64.3	90.0		.	7	1	2	2	1.9632	-10.0263
8	.	7	Unusual	Janet	F	62.5	112.5	Carol	.	8	1	2	2	0.1632	12.4737
9	.	8	Unusual	Mary	F	66.5	112.0		.	9	1	2	2	4.1632	11.9737
10	.	9	Unusual	Thomas	M	57.5	85.0		.	10	1	2	2	-4.8368	-15.0263
11	.	10	Unusual	James	M	57.3	83.0		.	11	1	2	2	-5.0368	-17.0263
12	.	11	Unusual	John	M	59.0	99.5		.	12	1	2	2	-3.3368	-0.5263
13	.	12	Unusual	Robert	M	64.8	128.0		.	13	1	2	2	2.4632	27.9737
14	.	13	Unusual	Jeffrey	M	62.5	84.0		.	14	1	2	2	0.1632	-16.0263
15	.	14	Unusual	Alfred	M	69.0	112.5		.	15	1	2	2	6.6632	12.4737
16	.	15	Unusual	Henry	M	63.5	102.5		.	16	1	2	2	1.1632	2.4737
17	.	16	Unusual	Ronald	M	67.0	133.0		.	17	1	2	2	4.6632	32.9737
18	.	17	Unusual	William	M	66.5	112.0		.	18	1	2	2	4.1632	11.9737
19	.	18	Unusual	Philip	M	72.0	150.0		.	19	1	2	2	9.6632	49.9737

Figure 3 shows a fairly complicated data step and illustrates many of the problems people have with the program data vector.

FIGURE 3

When the debugger starts up it adds two more windows to the display manager. If you look at Figure 4 you can see tabs for “debugger log” and “debugger source”. When the debugger is running, these are the two tabs that are used. Most users seem to arrange the windows as shown in Figure 4 below. The SAS code shows up in the debugger source window and SAS code can be many columns wide. People make the debugger source window wide to show all of the program code. The output in the debugger log has a vertical arrangement so most people tend to make that window narrow.

Debugger commands are entered in the place indicated by the yellow rectangle in Figure 4. As a warning, it can be hard to lay out the windows so that the complete area under the dash lines is visible. Often only the top half of the letters you enter into this “command” area can be seen and this makes entering commands difficult. The negative effects of this problem can be reduced by several techniques (cut – and – paste, binding commands to keys, macros, etc.) that are explained in the other DSD paper paper mentioned in the abstract.

The process of debugging is one where the programmer compares what he ***expects to see*** in the program data vector with ***what he actually sees***. The debugger allows a programmer to step through the code and watch the program data vector “execute”- and more.

The debugger also allows interactive debugging. It is easy to imagine, that a programmer might have an “a-ha” moment” and says something like “there is the problem. Right now variable X3 has the value of five and that 5 value is causing my problem.” The debugger allows the programmer to, manually, change the value of a variable (from 5 to some other number) and then execute more lines to observe if the new value of the variable “fixes” the problem. In this paper we will, manually, step through the program, one SAS statement at a time (st 1; ex _all_; <ret> and <f4><ret> and <f4><ret> and <f4><ret>). In practice, with large data sets, this can be impractical and the debugger has the ability to say “run my code and stop executing at one of several logically defined conditions occurs”. These features are not shown in this paper.

In Figure 4 we have started the debugger but not executed any statements. The black highlighted line shows the statement that ***will execute next.*** The black highlighted line will only stop on “executable statements”. Lines 415 to 417 are “compile statements” and the debugger had already processed those lines of code in the “compile” process – as we will see below.

NOTE: if a reader attempts to reproduce the examples shown in the paper it is to be expected that the line numbers in the debugger source will be different from the numbers shown in the paper. The line numbers that appear in the debugger source window are assigned by SAS and increase every time the debugger is started. If we were to shut down the debugger and immediately restarted it, the first line number would not be 415. It would likely be 435. While the numbers in your session may differ from the numbers shown in the screen prints in this paper, referring to specific lines by their line number is a useful communication tool for this paper.

A common, but possibly sloppy, phrase to describe how the data step works is the phrase “control” of the process. People tend to think of the black highlighted line, or maybe the line above it, being in “control” of the process because the lines are either executing or are about to execute. When people say things like “control went to the top of the data step” they mean that the black highlight went all the way to the bottom of the data step, hit the run statement, and then went back to the data statement (here line 415). Lots of important things happen to the PDV as the black line quickly moves through the data statement (line 419).

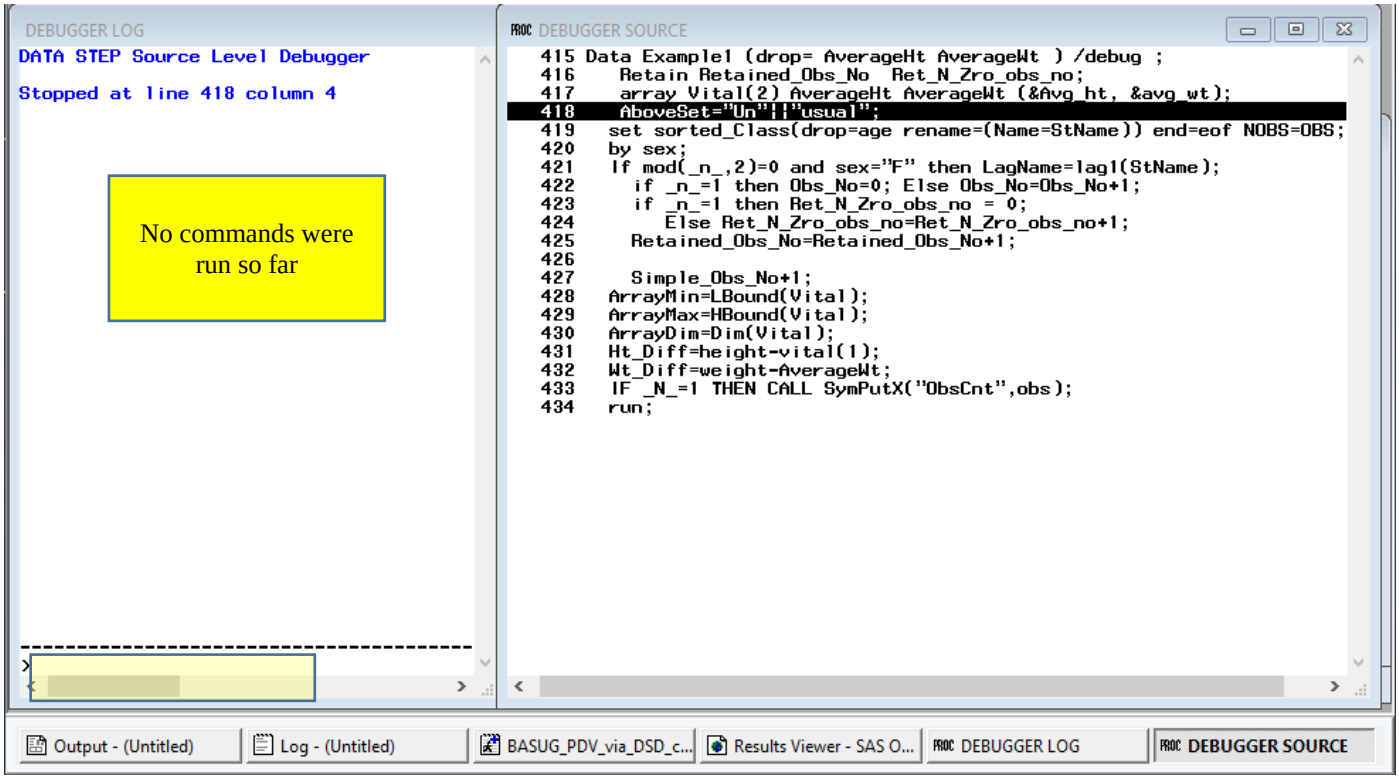

FIGURE 4

Debugger commands are entered into an area in the bottom of the debugger log. It is shown with the yellow rectangle in figure 4.

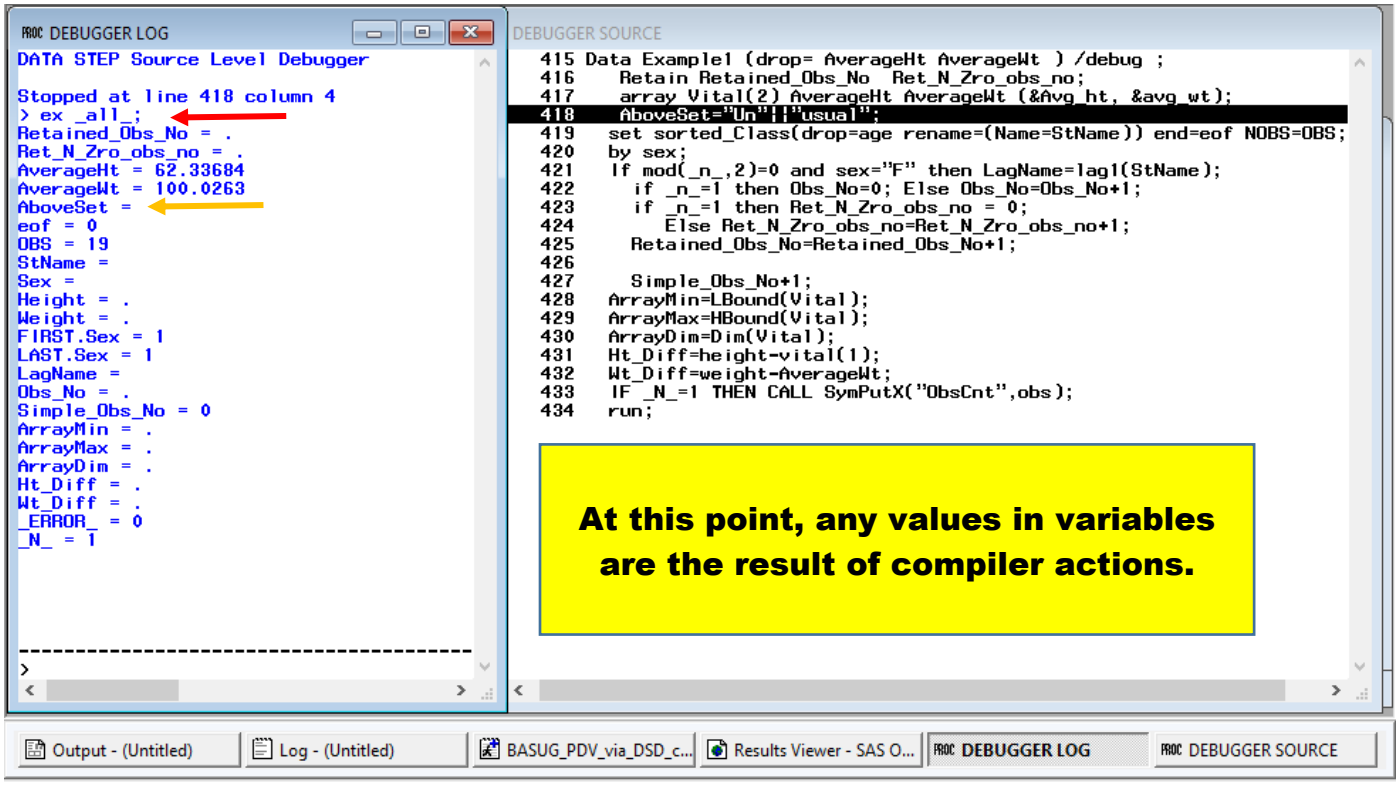

FIGURE 5

In figure 5, we have not executed any code. Line 418 is about to execute. We typed a debugger command “ex _all_; <ret>” to display the contents of the PDV in the debugger log. Here you can see the effects of “compile commands”; some variables are valued before data is read.

The macro variables were resolved during compilation and used to initialize the variables in the array statement. EOF (end of file) is valued as zero because we are not processing the last row of data in the data set. OBS has a value of 19. The compiler read the 19 value from the input table header. Below, you will find a bit of data _Null_ code that is a very fast way to get the number of observations in a data set into a variable or a macro variable.

Because the return of the number of observations in a data set happens in the compile step the code to right is a fast way of getting the number of observations in a data set into a macro variable. Because "if 0" is always false, the data set is not read, but the compile step does happen. The variable odds count is created and used in the call SymPutX..

```
Data _null_;
  if 0
/*this is false & NEVER executes*/
  then
   set SASHelp.class NObs=ObsCount;
 call SymPutX('ObsInFile', ObsCount);
run;

 %PUT 'OBSINFILE' = **&OBSINFILE**;
```

FIGURE 6

The values of sex in the data set are "F" and "M". Because we have no read any data, the current value of sex is missing. SAS knows that this is the only row where sex has a value of missing. Because there is only one row with sex equals missing the value of First.sex and Last.sex are both equal to 1.

The variable Simple_Obs_No is coded using the syntax '**SIMPLE_OBS_NO+1'.** This statement will increment the value of Simple_Obs_No by 1 every time that line of code is executed and has some other interesting effects. Specifically, the variable Simple_Obs_No will be initialized to zero without any special coding and will also be retained.

The variable _ERROR_ is set to zero because we have not encountered any errors in processing this loop through the data step. _N_ is set to one because we are in the middle of the first loop through the data step

Notice that the variables are created in the order in which the compiler "sees" the variables as it reads the code from top to bottom.

It is important to understand that we have not executed any SAS code in Figure 5. Figure 5 is about to process the first line of executable code (line 163). What we see in Figure 5 is the results of the SAS compilation step.

The array statement, online 162, created two new variables AverageHt and AverageWt. They were not in the data set SASHelp.class and were created during the compile step. One can also see that we initialized these two variables using macro variables and the macro variables were resolved to 62.33 and 100.02 during compilation.

The variable AboveSet is created by line 163 and line 163 is about to execute. Therefore, the variable AboveSet is missing.

On line 164 we coded in "end=eof" and "NObs=OBS". The command "end=eof" creates a variable called EOF that takes the value 1 when SAS is processing the last row of data in the SAS data set. "NObs=OBS" reads the header information from the input SAS file and finds the number of observations in that file – before any rows of data have been processed. The number of observations in a data set is often used as the denominator in some calculation.

The next four variables in the program data vector are: name, sex, height and weight. They all have the value of missing because the set statement has not executed. Data has not been read from the input file.

The variables First.sex and Last.sex both have the value 1 and deserve a short discussion. No data has been read and the value of sex is missing. SAS knows that the next observation will have a different value of sex and that this situation, where sex equals missing, will only happen for "one row". If your by variable produces a group that has only one row, the variables First.sex and Last.sex ***both*** have the value 1 for that row of data. These values will change as soon as the set statement executes. Ht_diff and Wt_diff will be not be valued untill we execute lines 166 and 167. .

ERROR is a temporary SAS variable (it is not sent to the output data set) and has a value of zero because no errors have been found, so far, in this " loop through the data step". _N_ has a value of 1 because this is our first "loop through the data step".

Please see red the arrows in the figure below. You can string commands together inside the data step debugger command line by separating them with a semicolon;.

The command below will cause the debugger to execute one line of code and then display all of the values that are in the program data vector.
"step 1; ex _all_;" (you can easily re-issue the command with <F4><ret>)
Please note that the second semicolon is important in making this command run as desired.

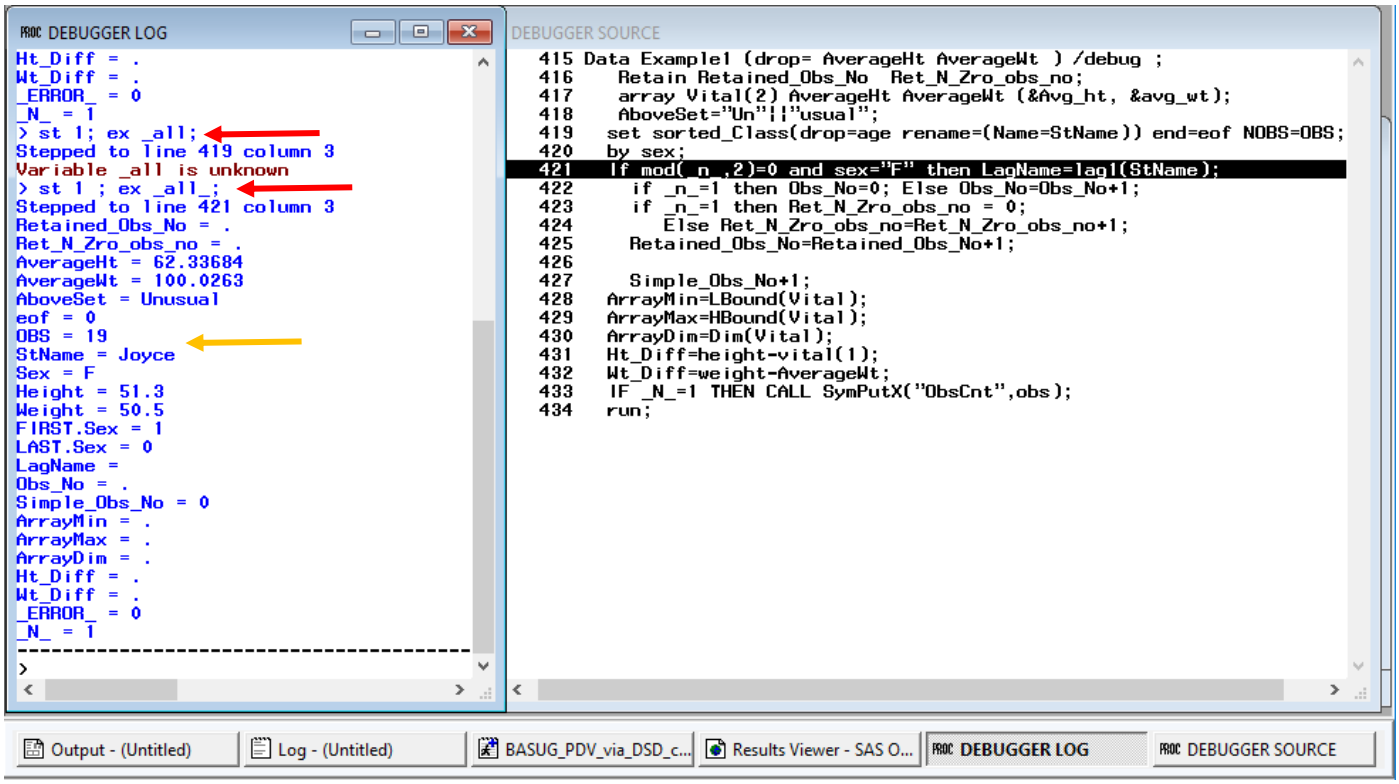

FIGURE 7

Lines 418 and 419 executed (note the error in the debugger log) and 421 is about to execute. The red text in the debugger log in Figure 7 shows that I made a typing error. Instead of typing "st1; ex _all_;" I typed "st1; ex _all;"

The debugger tried to execute my two-part command and had a problem. It executed line 418 (AboveSet="Un"||"usual") – but the display part of the command did not happen.

I then reissued the correct command (st1; ex _all_;). In Figure 7 you can see that the variable AboveSet= has the value "Unusual". The set command (419) has executed and values were put into the PDV. Please see that the values in the debugger log correspond to values that are shown in Figure 2.

The row we are processing is the first row where the variable sex has the value "F". Accordingly First.sex has the value of 1. There are more females yet to be processed and so the value of Last.sex is 0.

The line in Figure 5 that was about to execute was the line '`ABOVESET="UN"||"USUAL";`'

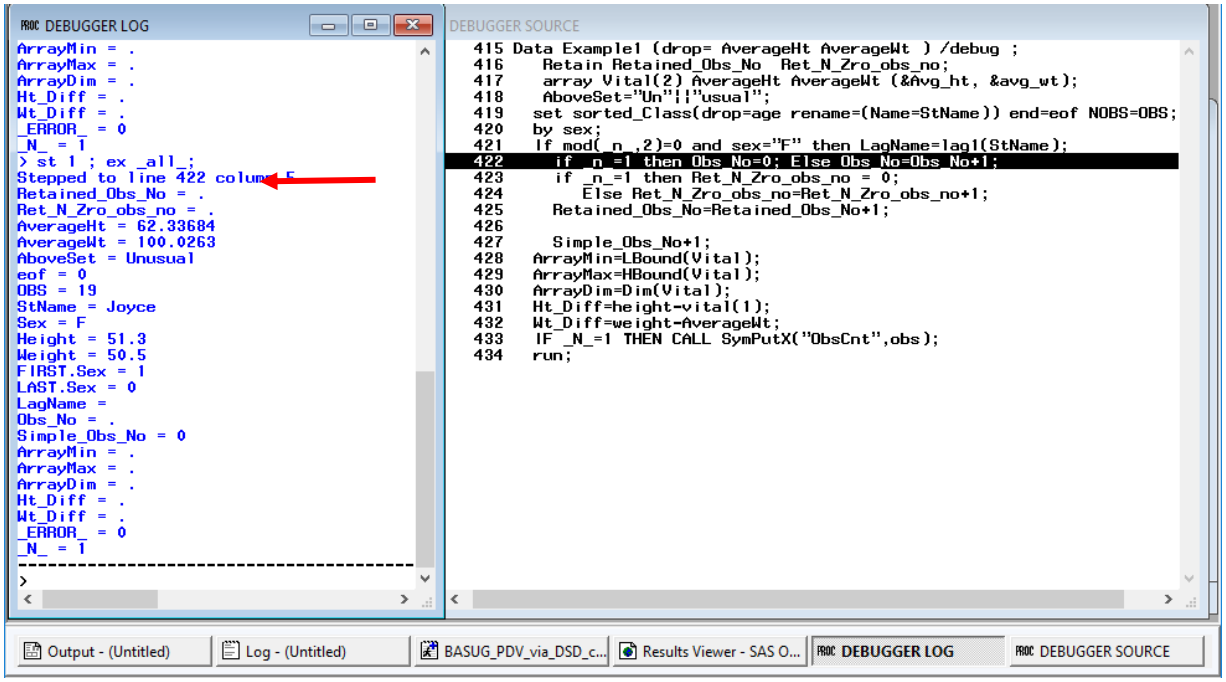

FIGURE 8

lLne 421 executed and 422 is about to execute. Several kinds of SAS statements are executed in multiple steps. Line 421 executes in multiple steps. The first step will evaluate the mod function, the second will evaluate the sex equals function and finally the lag statement will execute. Since this is the first time the lag has executed, LagName is assigned a value of missing.

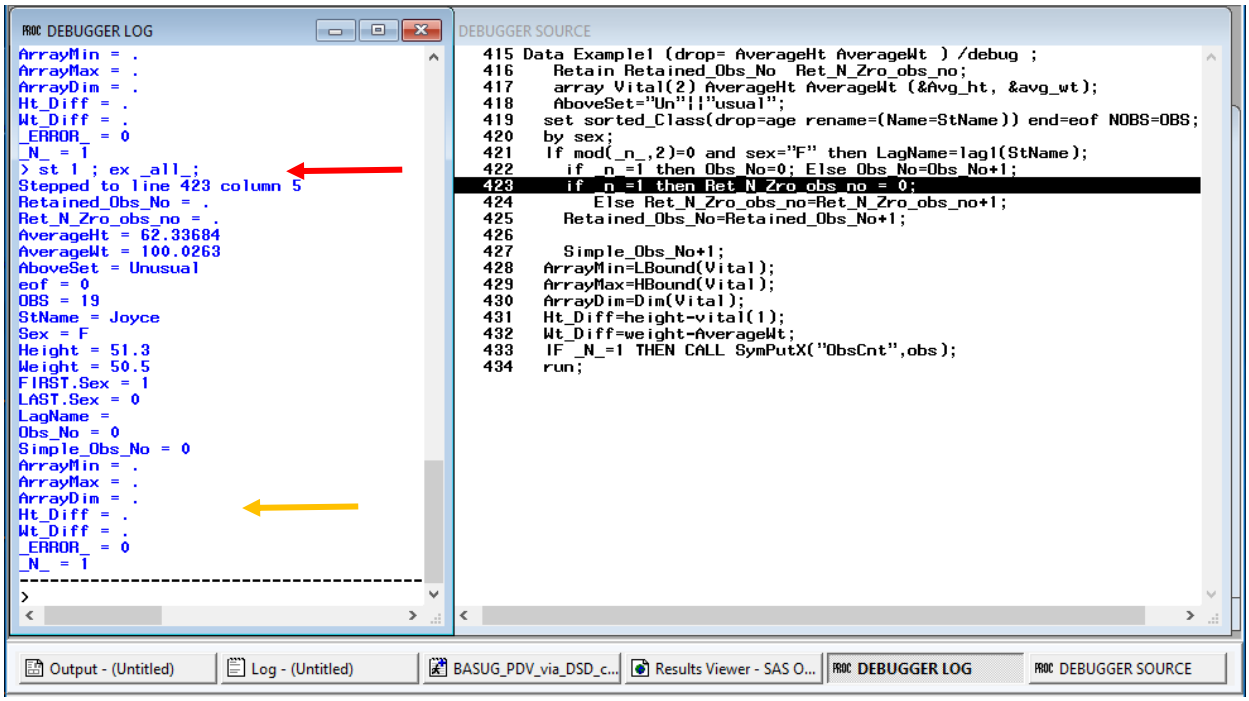

FIGURE 9

Line 422 executed and 423 is about to execute. Line 422 assigned a zero to the variable Obs_no. This assignment is an initialization of a variable to zero. It was done because Obs_no was created with a value of missing and SAS will later think that missing +1 equals missing. The assignment of a value to Obs_no is a “regular SAS assignment” (though part of an if else statement). The value of Obs_no will not automatically be retained and we have not explicitly requested that this variable be retained. (This is a logic error. The variable was intended to be an observation counter and will now be off by one)

Getting the debugger to “move to/highlight” line 425 will take multiple issuances of <f4> <ret>

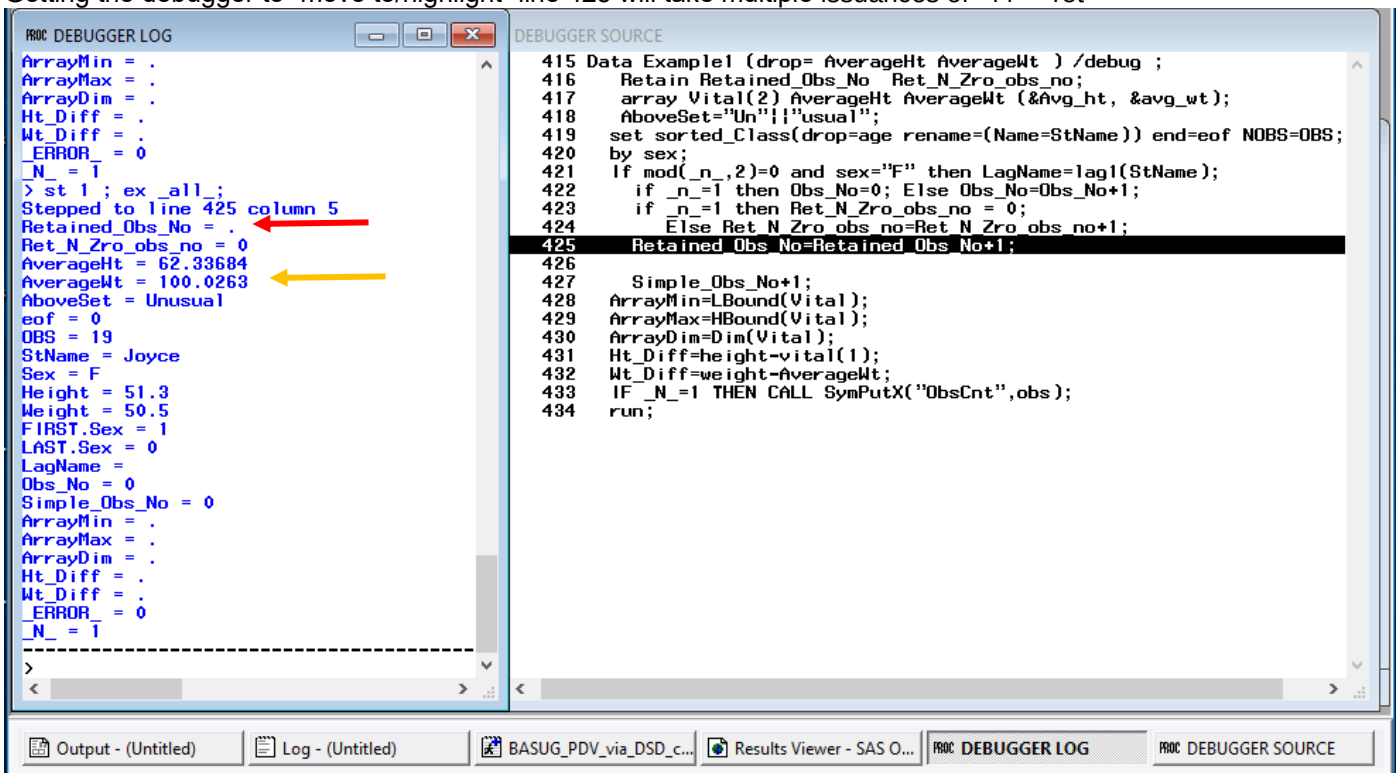

FIGURE 10

Lines 423 executed and line 424 was “skipped”. 425 is about to execute. Remember that lines 423 and 424 are linked in their logic and their execution. Since _N_ is 1, line 423 executed and controlled jumped over line 424.

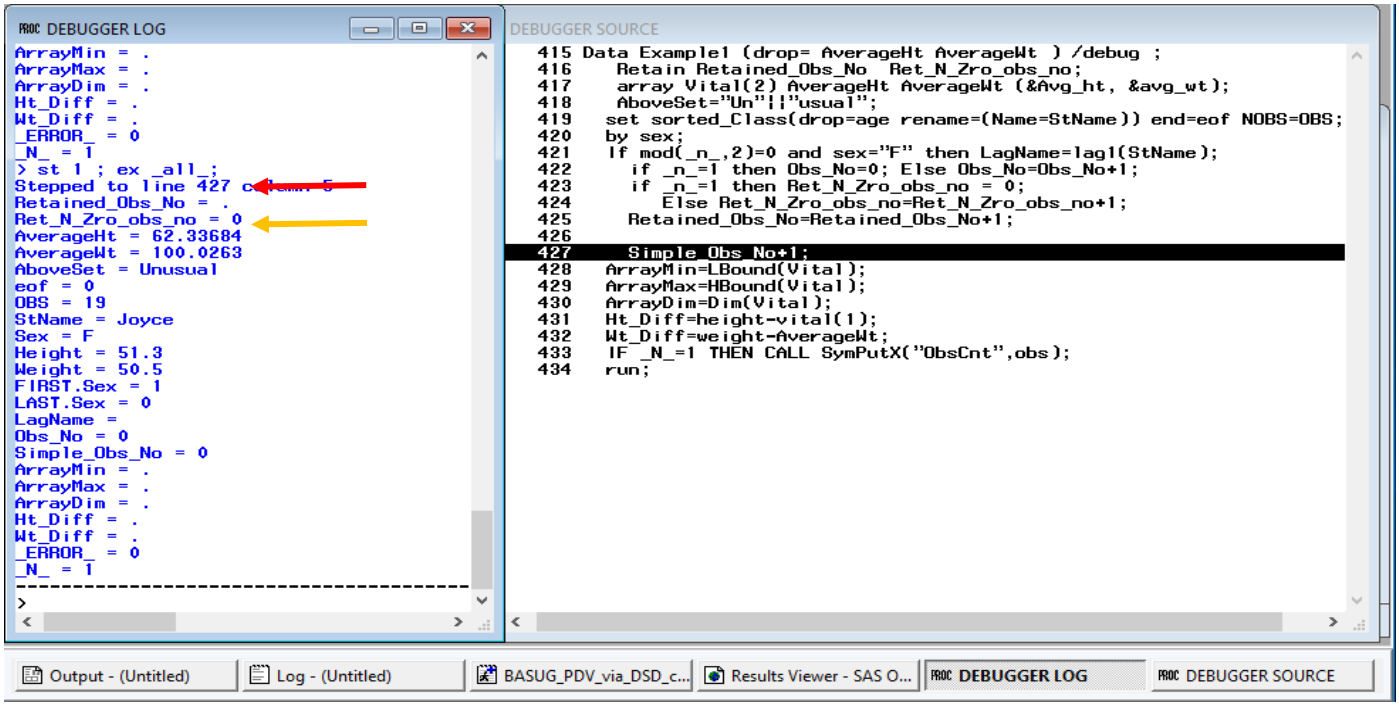

FIGURE 11

Line 425 executed and 427 is about to execute. Line 425 added 1 to the value of the variable Retained_Obs_No. Retained_Obs_No had been valued as missing and SAS documentation tells us that missing plus one equals missing. Note that the code “Retained_Obs_No. = Retained_Obs_No. +1;” is different from “Simple_Obs+1;” .

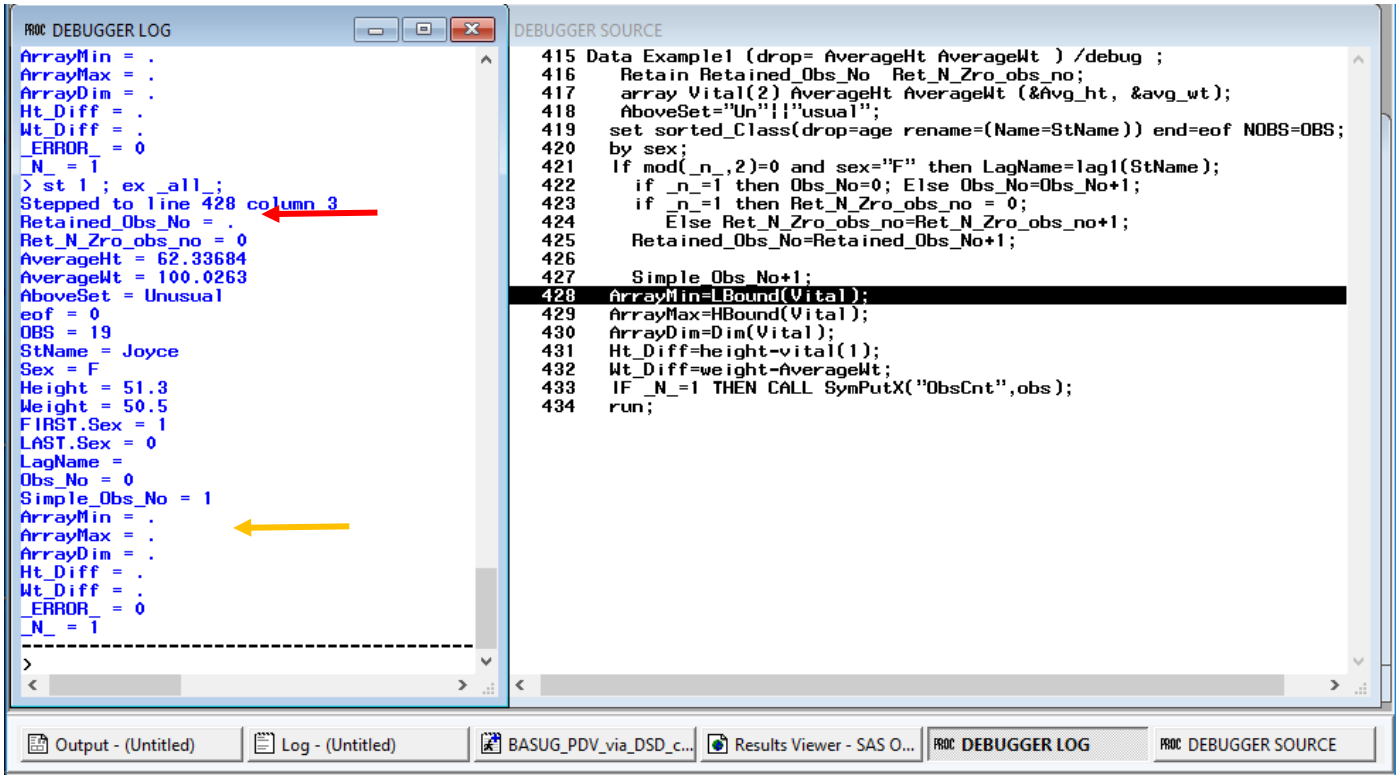

FIGURE 12

Line 427 executed and 428 is about to execute. Line 427 added one to the value of the variable Simple_Obs_No. Simple_Obs_No had been valued as 0 and now is valued as 1. Because of the syntax (Simple_Obs+1;) used to create this variable it is automatically initialized to 0 and also automatically retained.

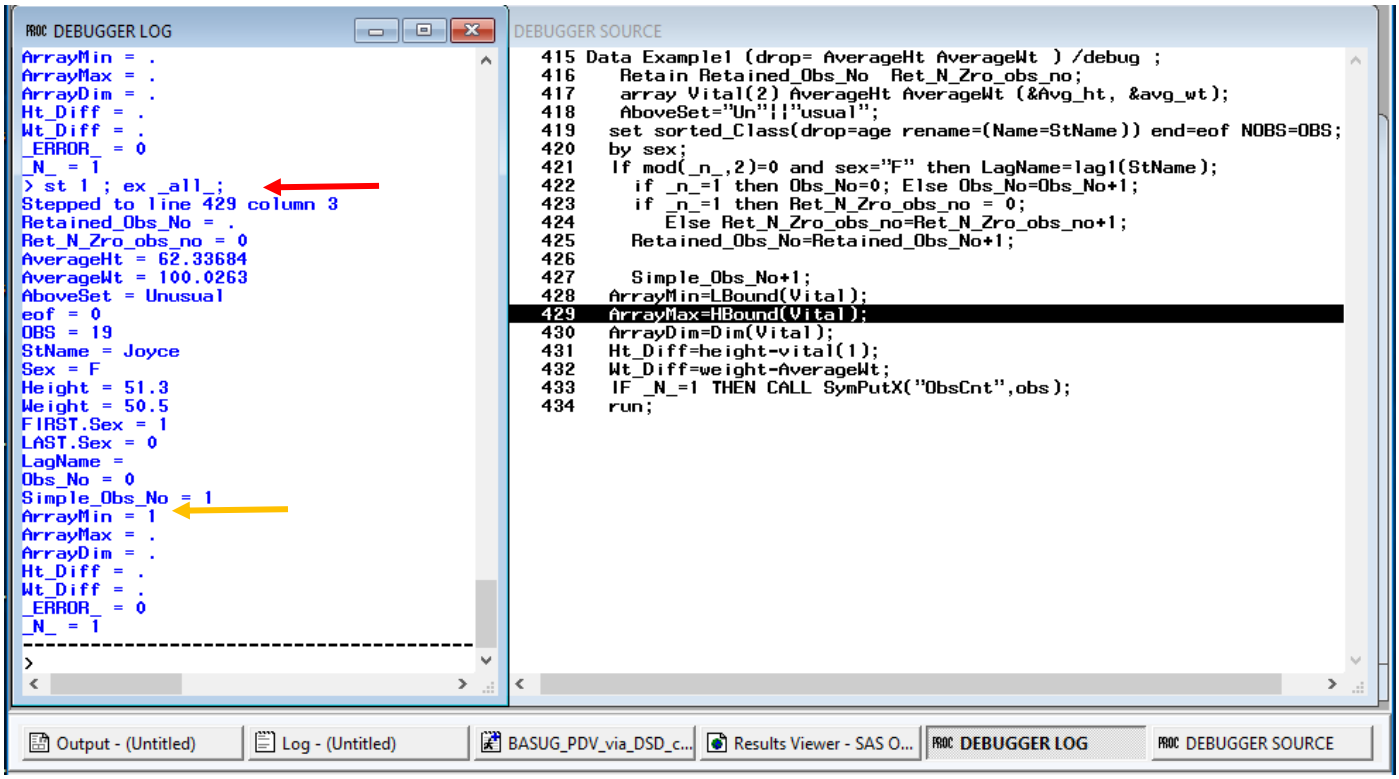

FIGURE 13

Line 428 executed and 429 is about to execute. Line 428 used a SAS function that “finds” the lower bound of an array. Here we loaded the lower bound value into the variable ArrayMin. This is an unusual use for LBound. LBound is normally used for looping through the elements of an array.

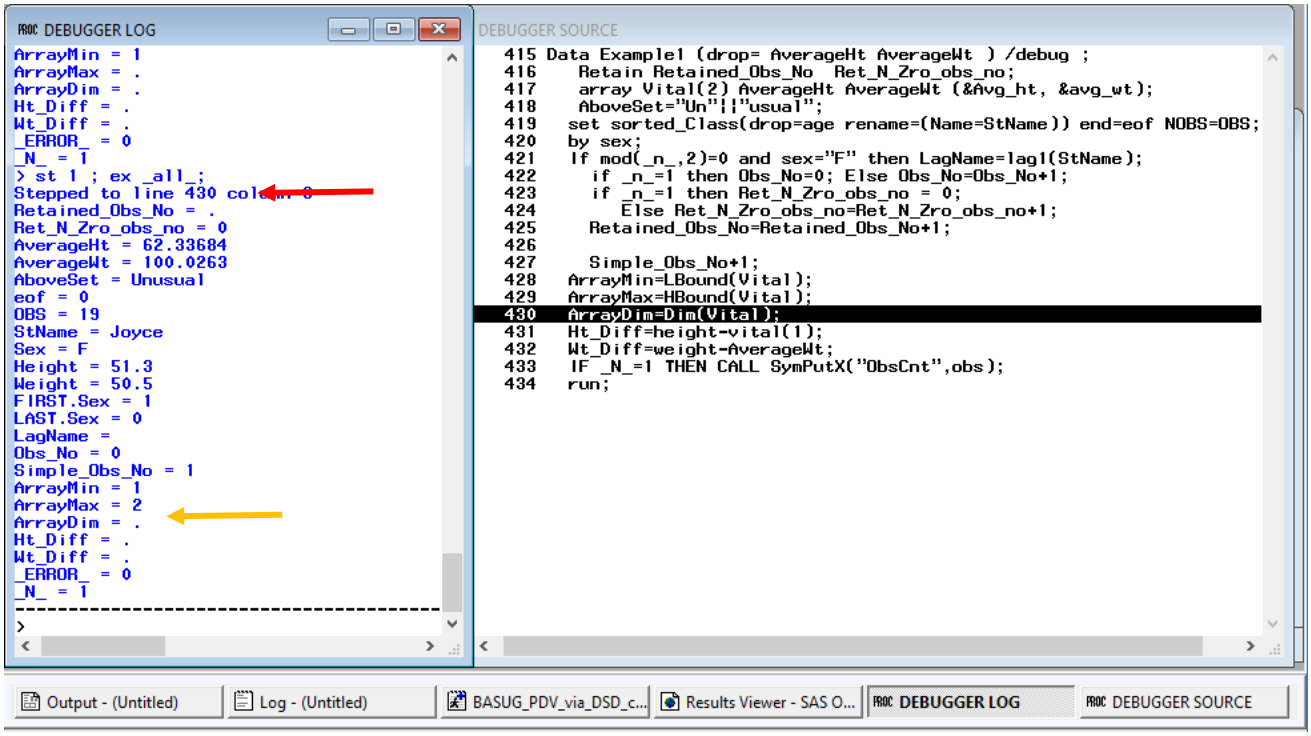

FIGURE 14

Line 429 executed and line 430 is about to execute. Line 429 used a SAS function that "finds" the upper bound of an array. Here we loaded the upper bound value into the variable ArrayMax. This is an unusual use for HBound. HBound is normally used for looping through the elements of an array.

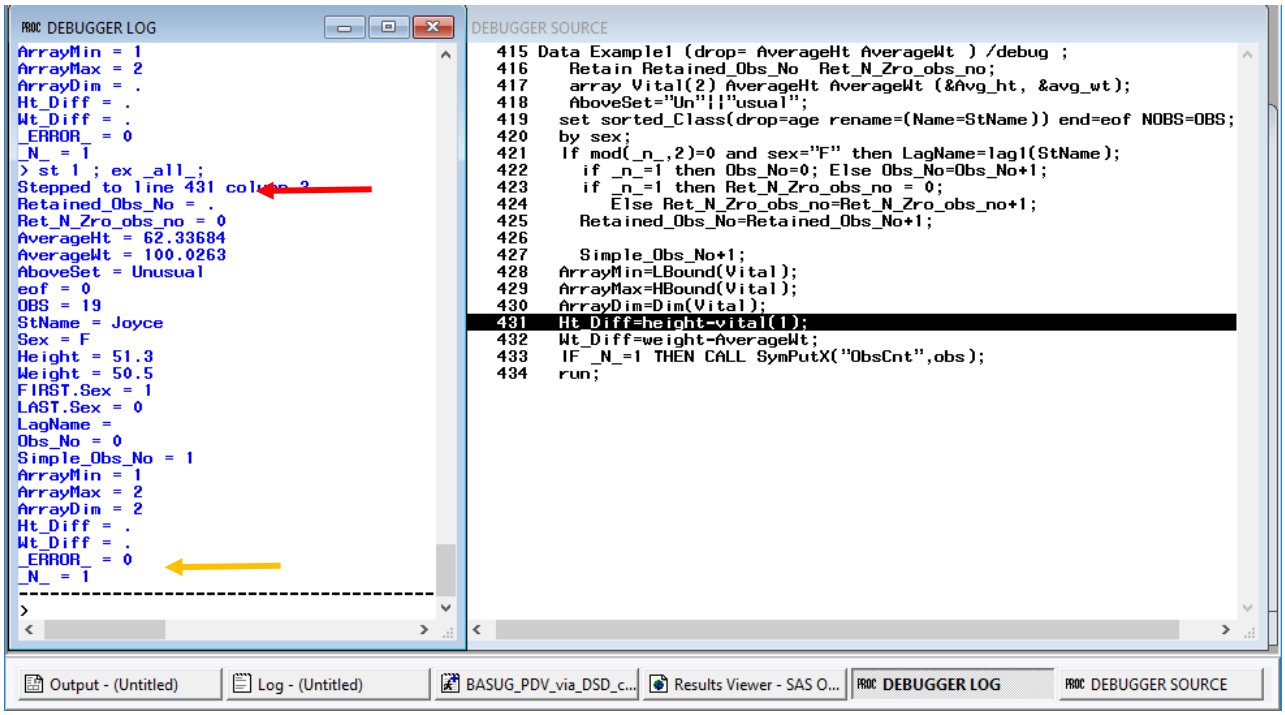

FIGURE 15

Line 430 executed and line 431 is about to execute. Line 430 used a SAS function that "finds" the dimension (the number of elements) of an array. Here we loaded the number of elements in the array into the variable ArrayDim. This is an unusual use for Dim. Dim is normally used for looping through the elements of an array.

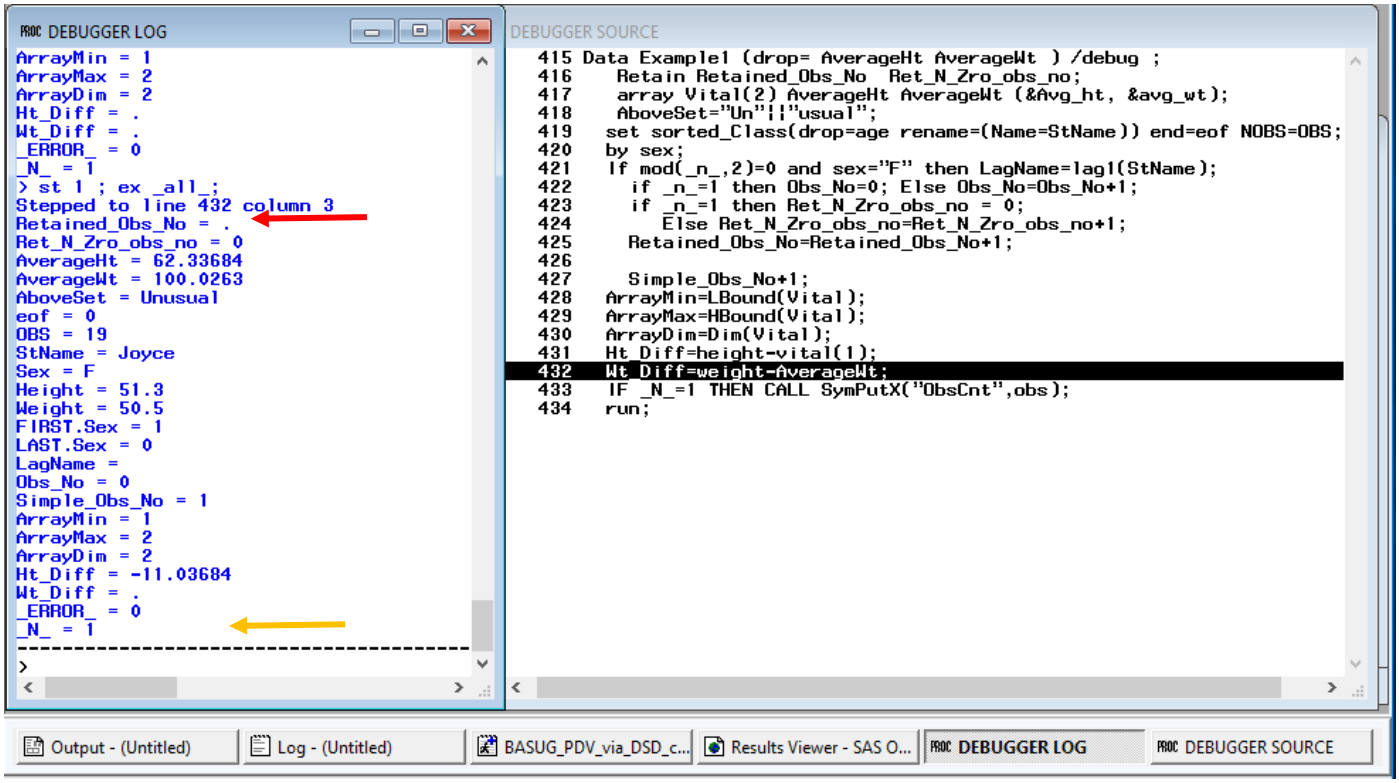

FIGURE 16

Line 431 executed and line 432 is about to execute. Line 431 used one of the two ways to access an array element to calculate a student's difference in height from the class average.

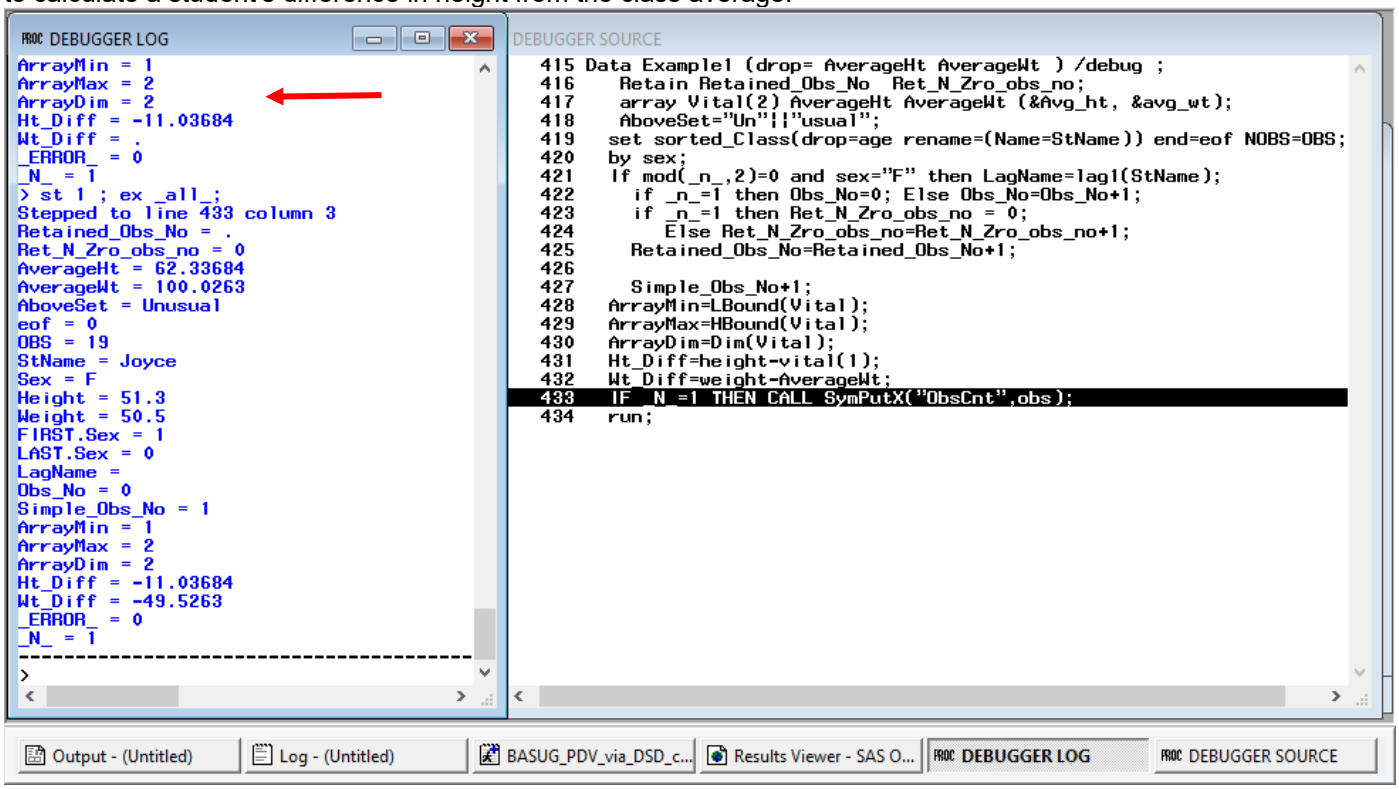

FIGURE 17

Line 432 executed and line 433 is about to execute. Line 432 used one of the two ways to access an array element to calculate a student's difference in weight from the class average.

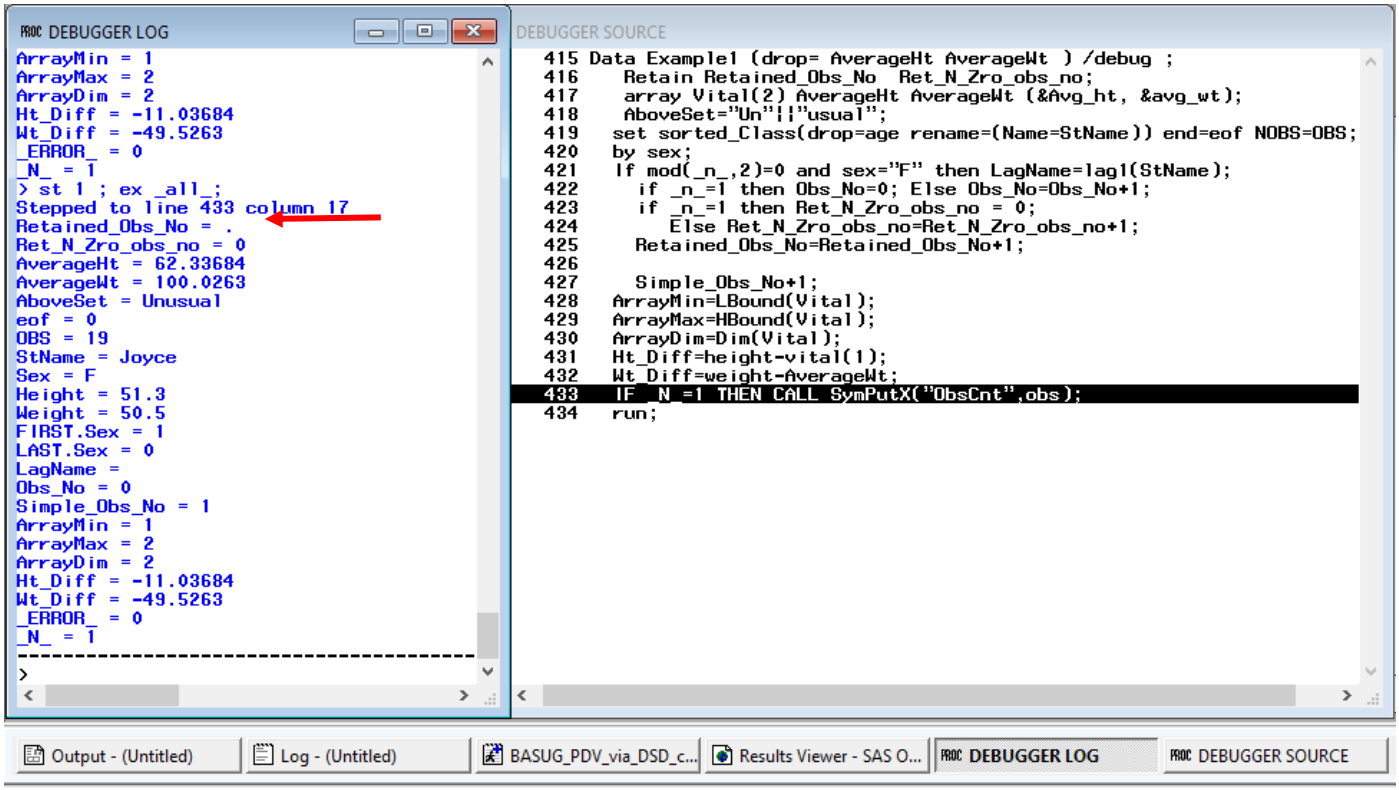

FIGURE 18
Line 432 executed and line 433 is about to execute – in multiple steps. You might have to issue multiple <F4><ret> commands to move off this line.

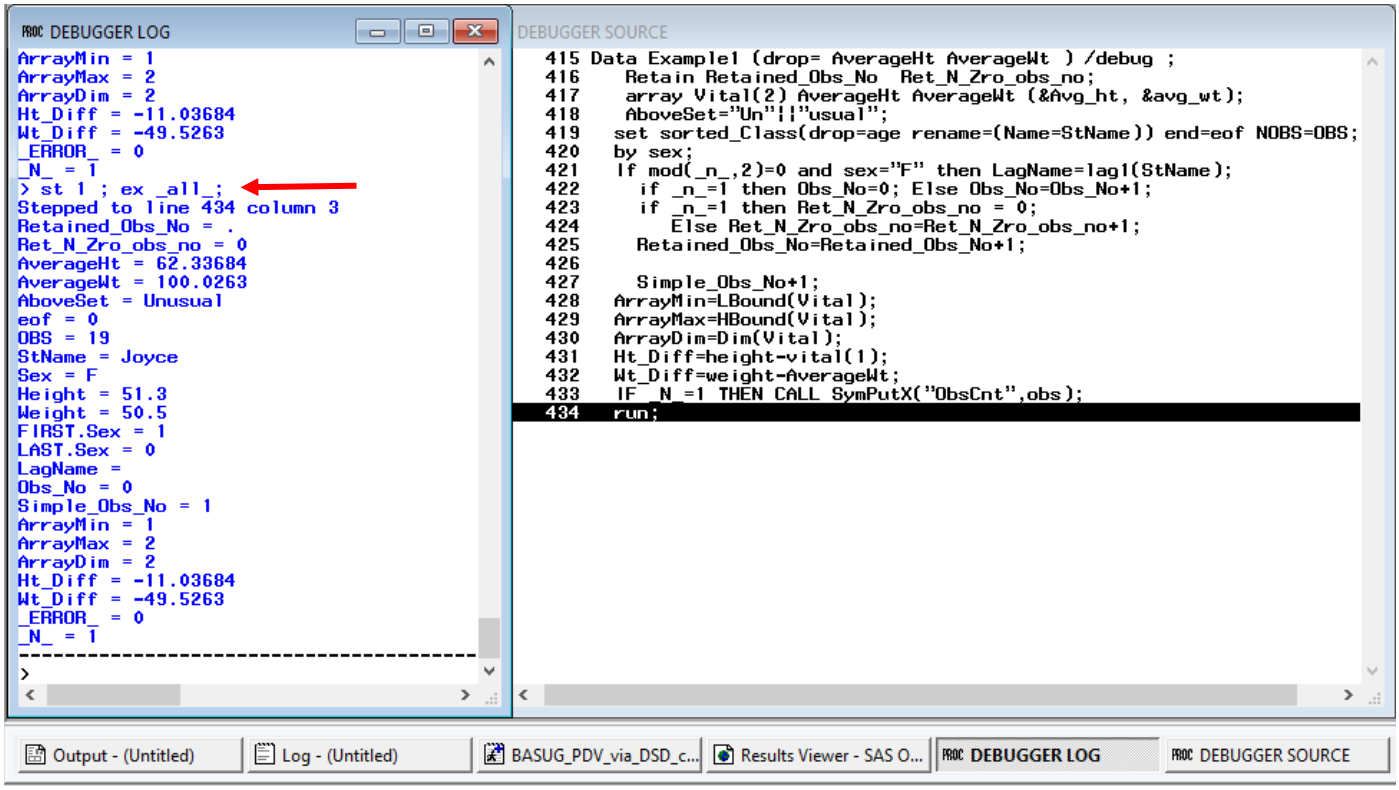

FIGURE 19
Line 433 executed (or is executing) and line 434 is about to execute. You might have to issue multiple <F4><ret> commands to get the black hi-lighting to move off this line.

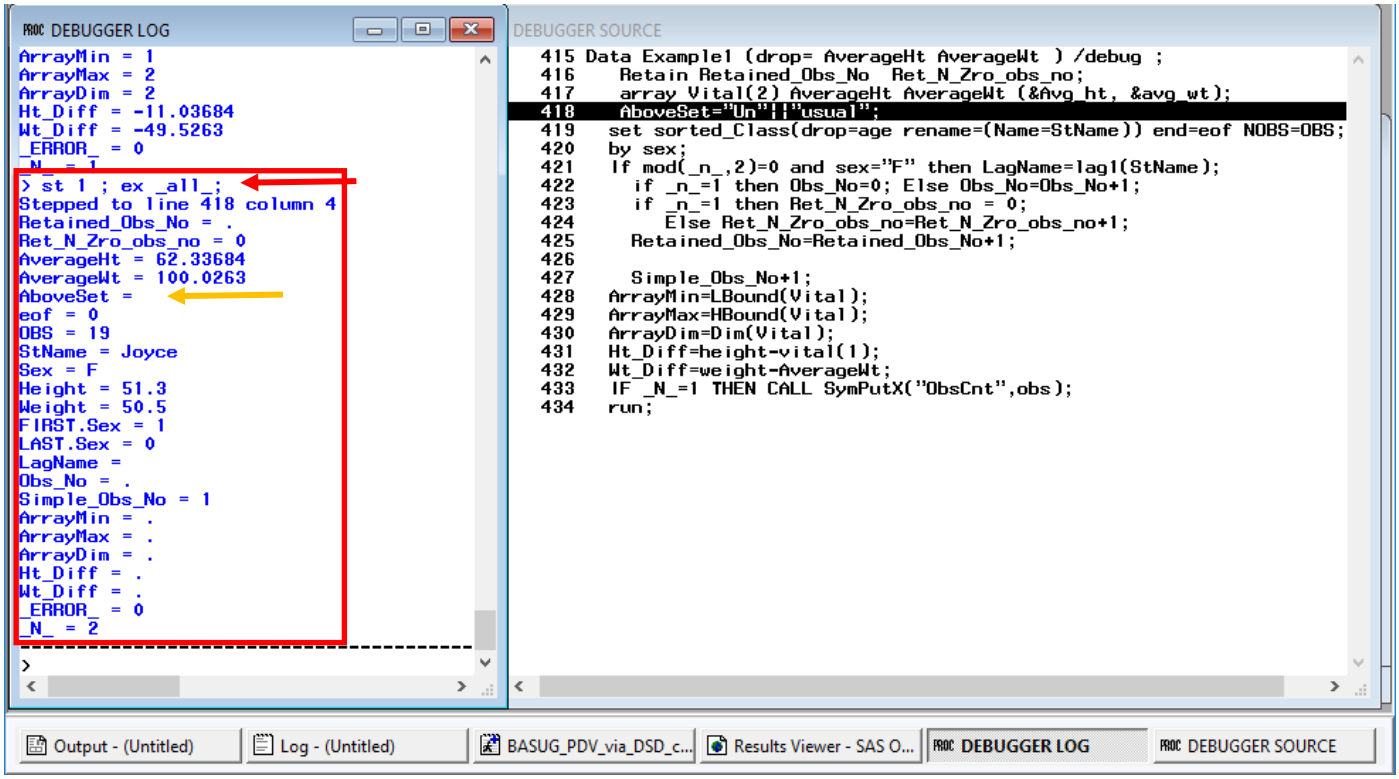

FIGURE 20

Line 434 executed and control of execution looped back to the top of the data step (line 415) and then down to the first executable statement which is line 418. Lines 415 to 417 are compiler instructions and some compiler instructions executed when control passed through 415. Line 418 will execute next.

Let's take this time to see what happens to the program data vector when control goes to the data step line (415).

_N_ now has the value of 2 because we are in our second "pass" through the data step. _ERROR_ is 0 because we have not found any execution errors. _Error_ is reset to zero as control passes through the data step line.

Variables that were created through assigned statements have been reset to missing (AboveSet, ArrayMin, ArrayMax, ArrayDim, Ht_Diff, Wt_Diff) as control passed through the data statement (line 415).

You can see the result of a logic error here. I retained Ret_N_Zro_obs_no and then I typed
if _n_=1 then Ret_N_Zro_obs_no = 0; (I wanted this to be an obs/loop counter)
Else Ret_N_Zro_obs_no=Ret_N_Zro_obs_no+1;

Since this variable is intended to be a row counter, I should have typed
if _n_=1 then Ret_N_Zro_obs_no = 1;
Else Ret_N_Zro_obs_no=Ret_N_Zro_obs_no+1;

Notice that the variables read using the set statement ***have been retained*** in the PDV. _N_ is 2, but the PDV contains information from the first observation (Joyce) in the data set. Please note that the current values for First.sex and Last.sex are appropriate for the first row of data to be read from the data step.

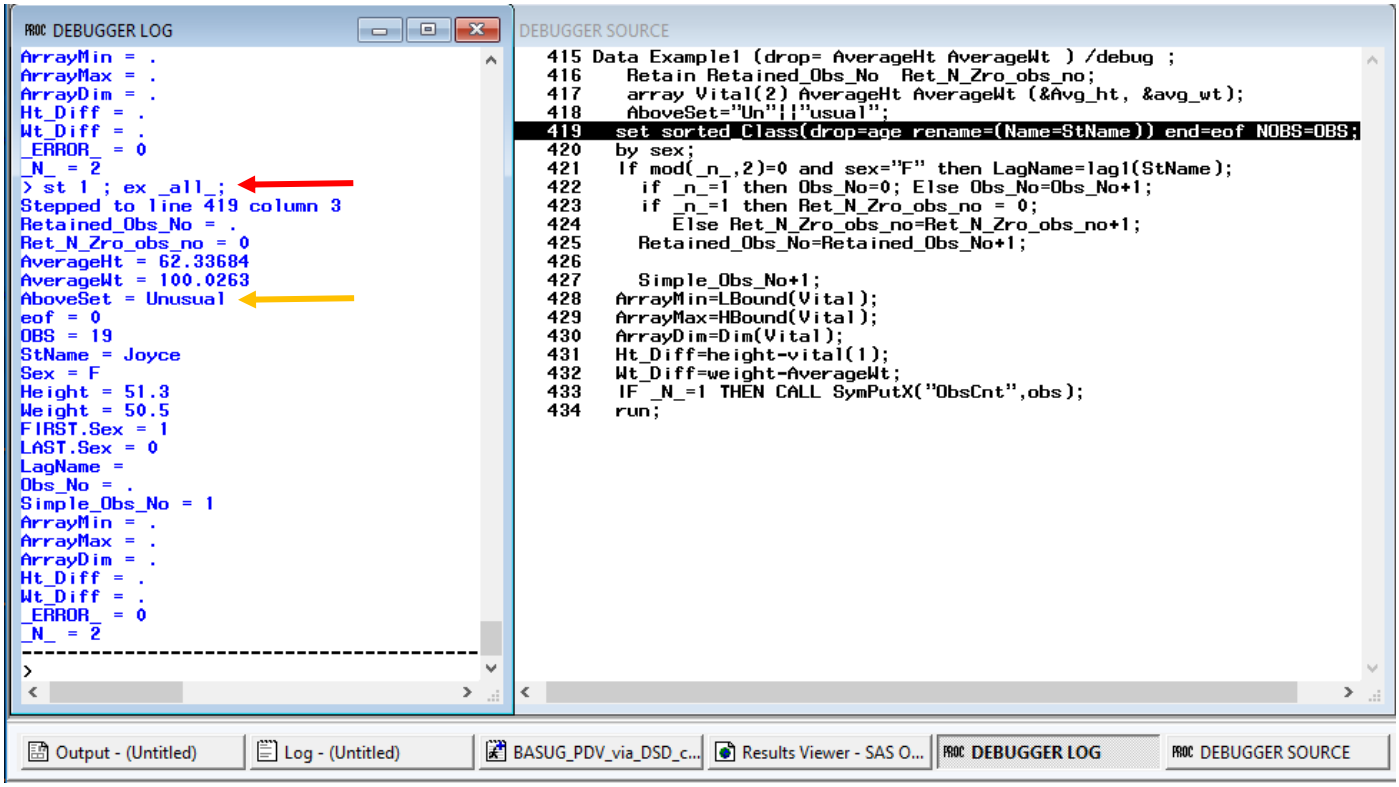

FIGURE 21
Line 418 executed and Line 419 will execute next. AboveSet has the value “Unusual”. We are about to read data.

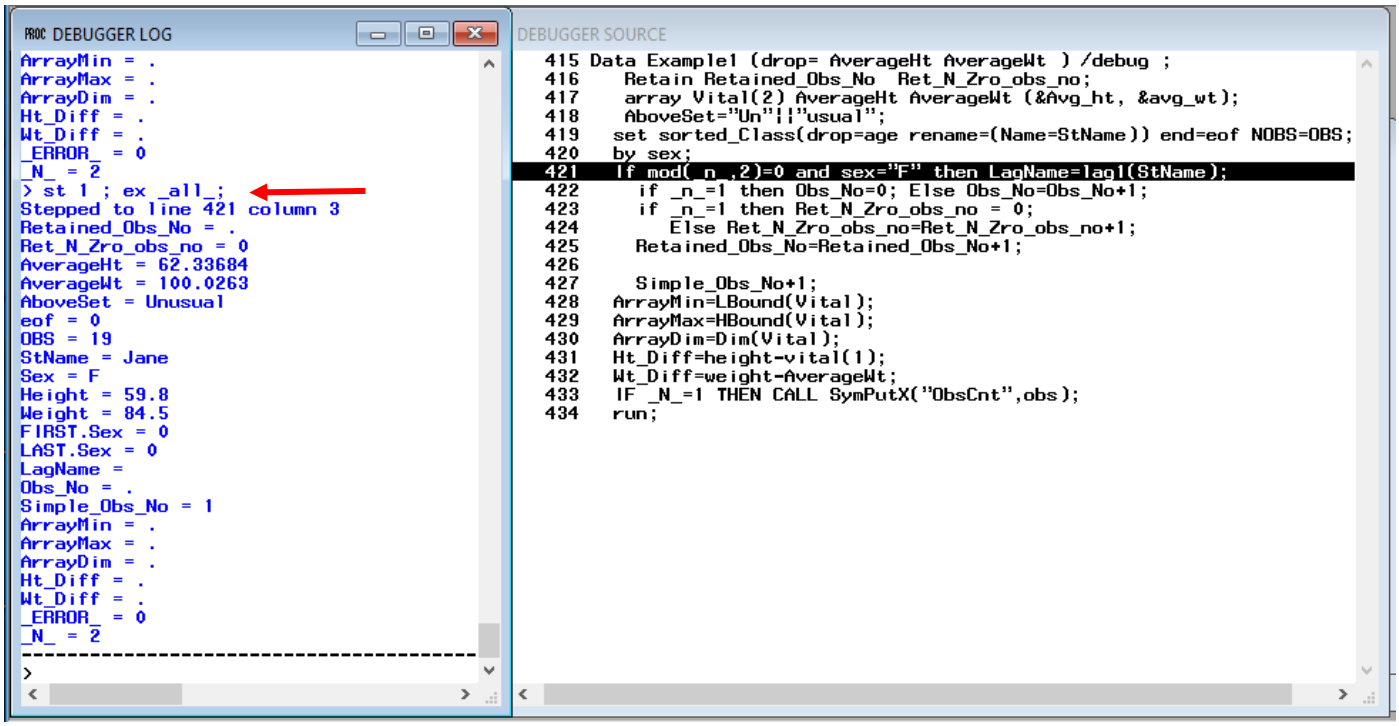

FIGURE 22
Line 419 executed and, since line 420 is a compile line, it is “jumped”. Line 421 will execute next. Line 419 brought in a new observation. Let’s look at the effect of the new observation on the PDv.

The student information (name, sex, height, weight) has changed. First.sex and Last.sex are both valued at zero.

You might have to issue <F4><ret> a few times to get the highlight to move off of line 421.

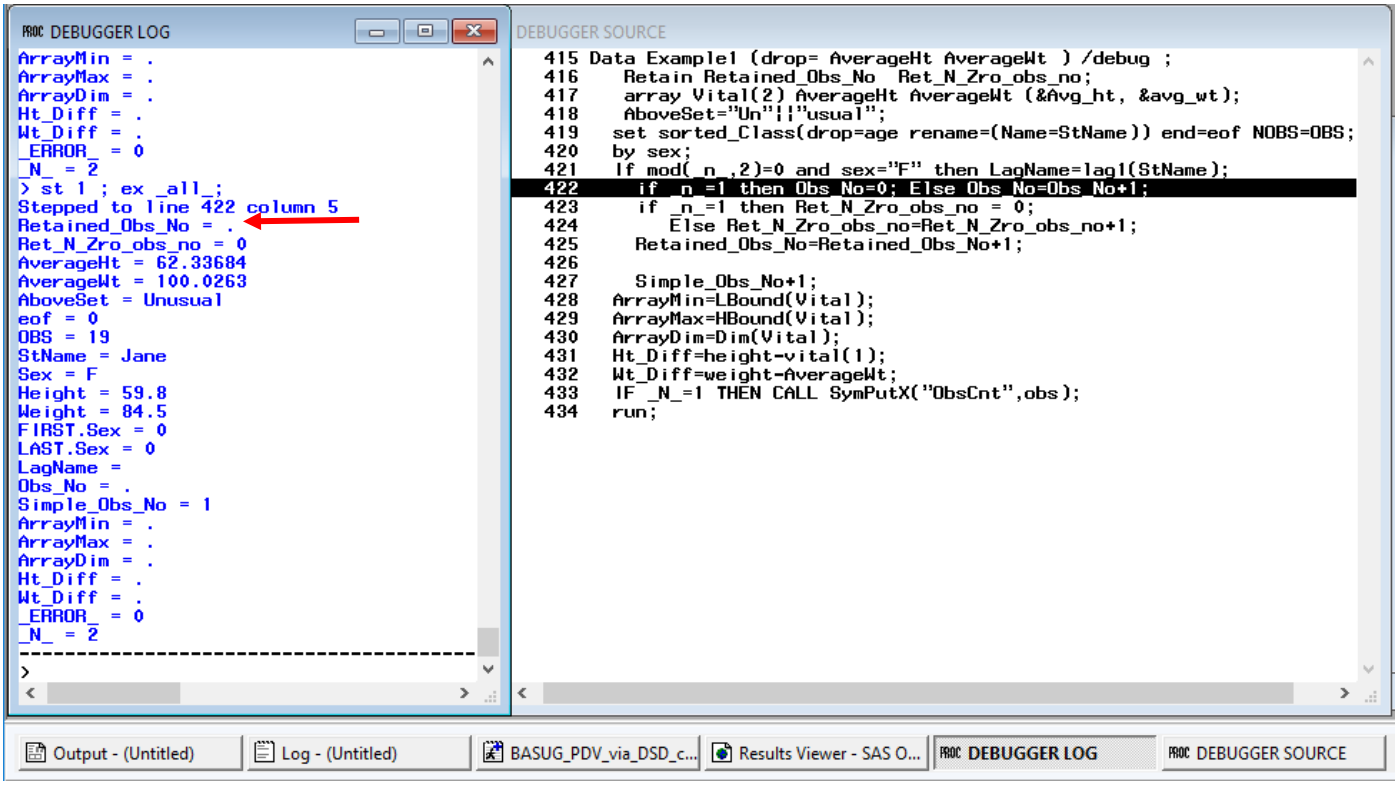

FIGURE 23

Line 421 executed and line 422 will execute next. Line 419 brought in a new observation. The student information (name, sex, height, weight) has changed. Sex is “F” and Mod(_n_,2)=0 is true so the lag executes for the first time, but the variable has no value.

The lag function does not return the last value of StName in the input data set. It returns the last value of StNname **from when the lag function last executed**. The lag is implemented through a queue that is in RAM and separate from the PDV. Since this is the first time that mod(_n_,2)=0 and sex="F" are both true, this will be the first time that the lag function executes. Jane will be the first girl to have her name put into the lag queue. When the lag function next executes, regardless of how many rows of data from the input data set have been processed between the queue loading and unloading, the string “Jane” will be returned.

You might have to issue <F4><ret> a few times to get the highlight to move off of line 421.

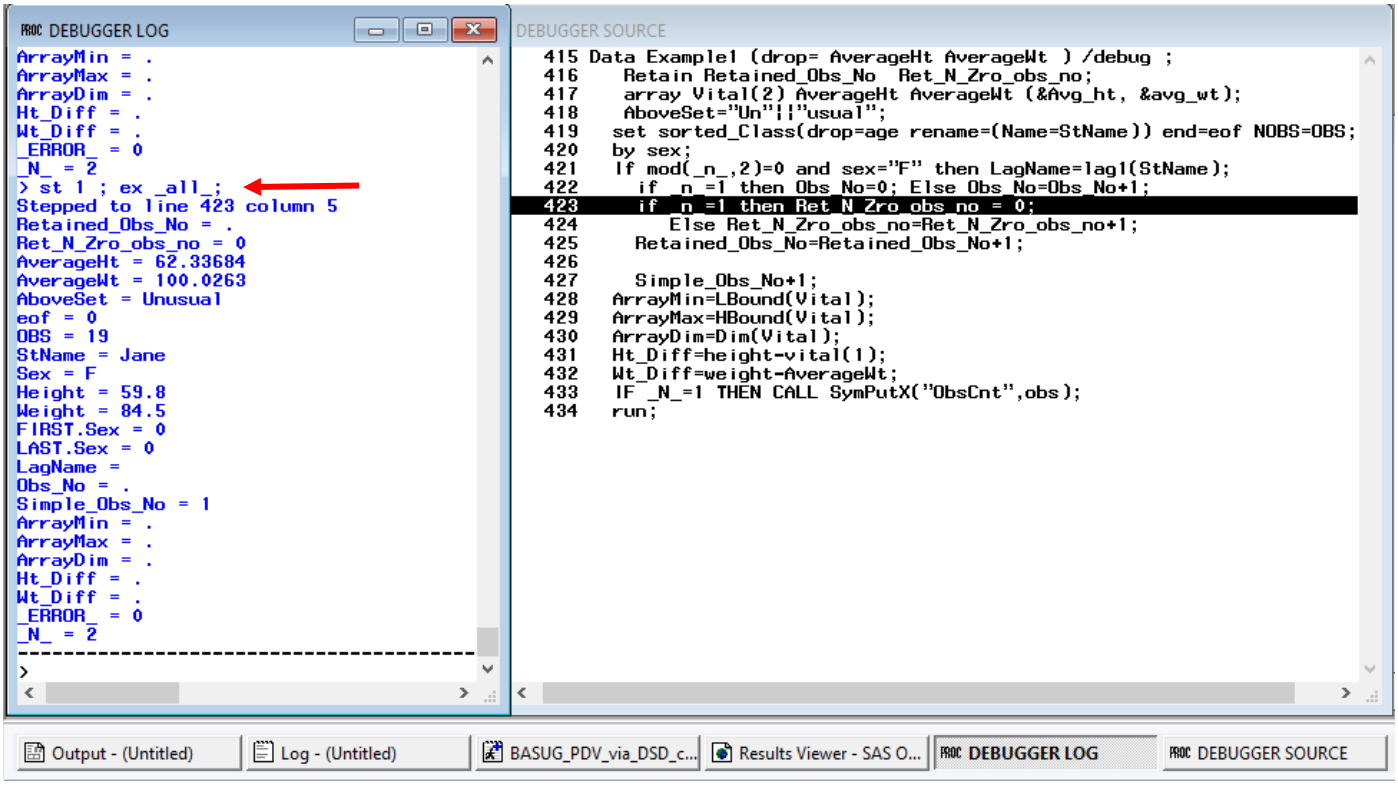

FIGURE 24

Line 422 executed and line 423 will execute next. On line 422, we had initialized Obs_No to zero and line 422 tried to increment the value in the variable Obs_No. While we had initialized Obs_No to zero, we did not retain Obs_no and it was reset to missing when control reaches the top of the data step (415). Missing plus 1 equals missing, so Obs_No is valued as missing..

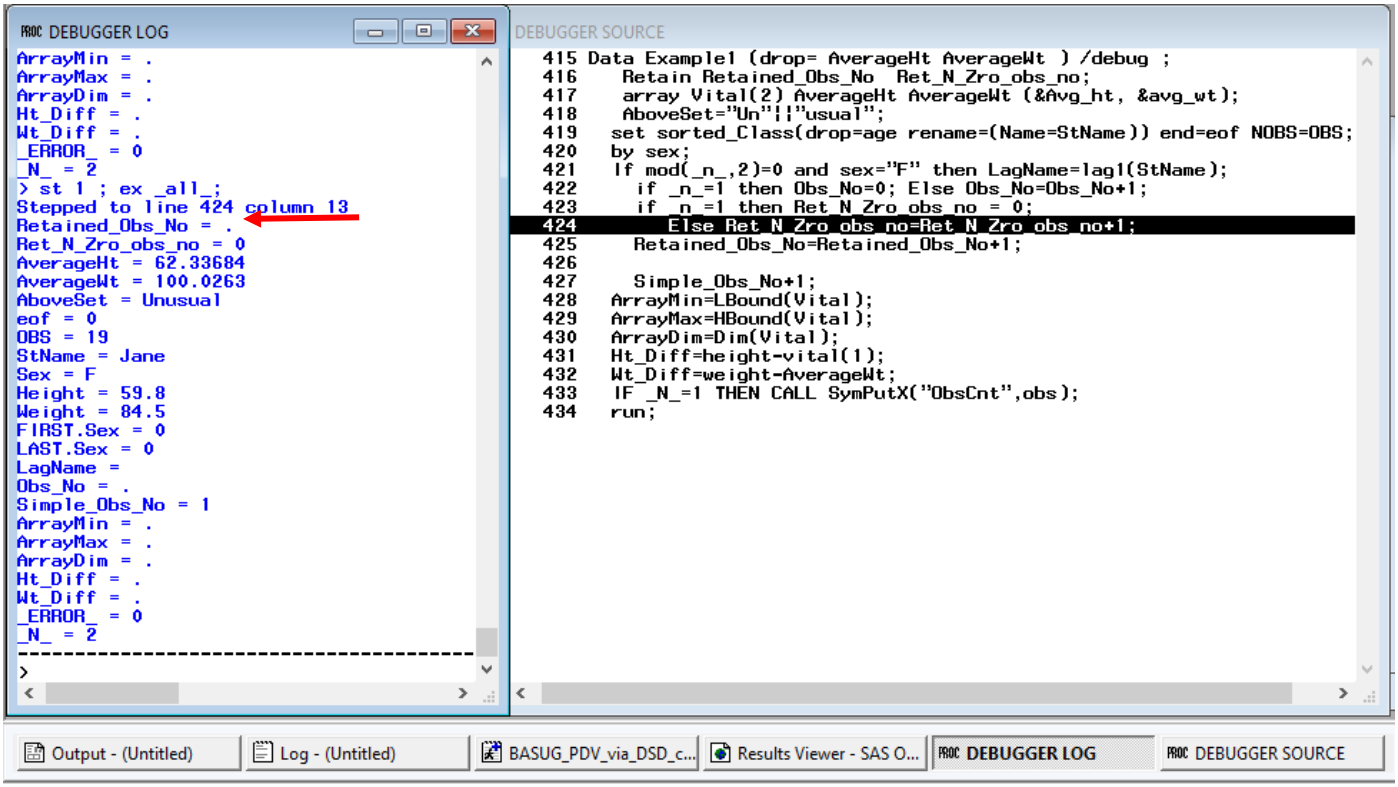

FIGURE 25

Line 423 executed and line 424 will execute next. Line 423 only executes when _N_ has a value of one. It does not execute in this “pass” through the data step.

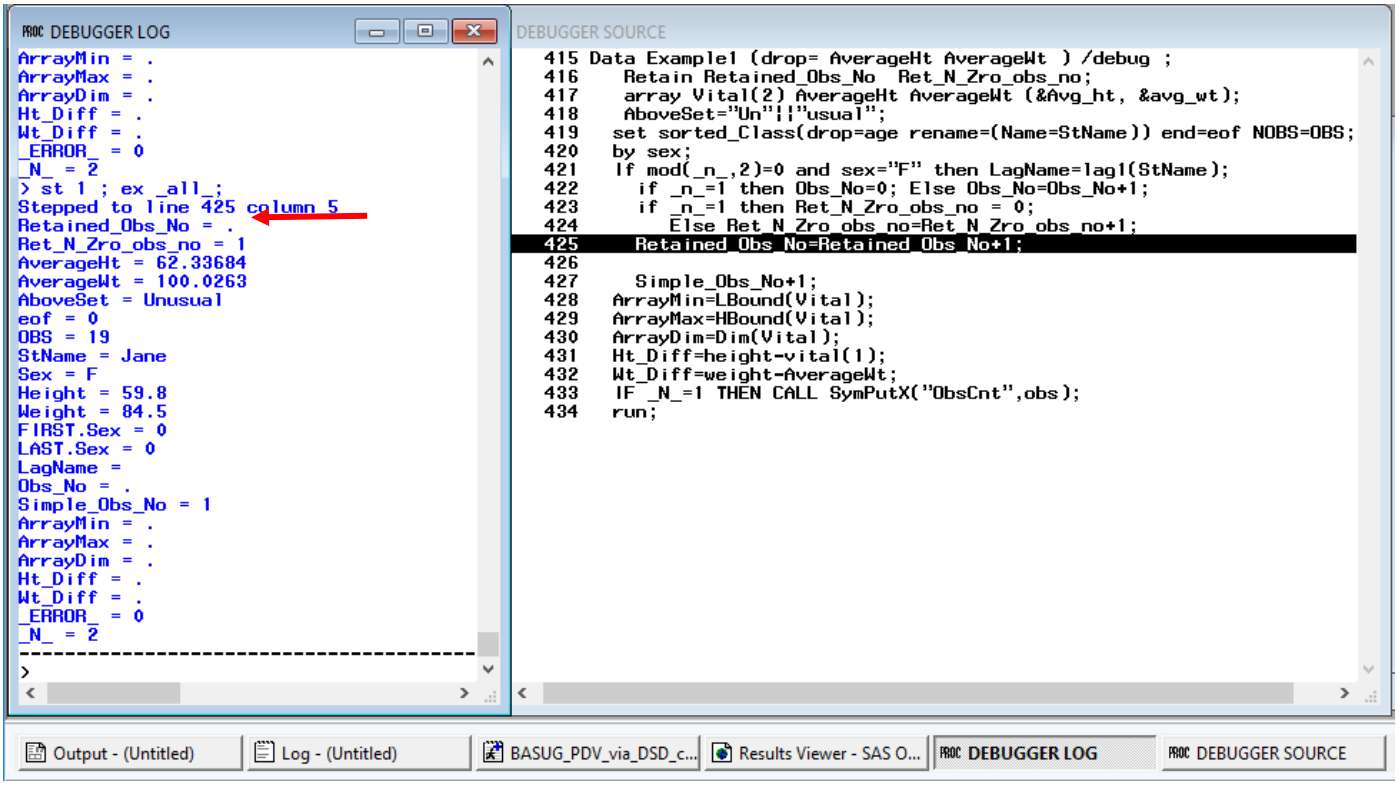

FIGURE 26
Line 424 executed and line 425 will execute next. Line 424 added 1 to the value in Ret_N_Zro (a variable that we had initialized to zero **and** retained). This variable now has the value of 1. Remember; this is the variable on which I'd made a logic mistake. We would all expect this variable to have a value of 2 at this time.

EXAMPLE 1 SUMMARY

At this point, a reader might want to go back to Figure 3, which shows the complete output data set. A good test of PDV comprehension would be to see if all of the values in the variables now make sense. The lag function is particularly tricky.

EXAMPLE 2

In this example we will see the unexpected effect of the automatic retaining that happens when we do a one to many merge and a calculation is performed on a variable in the “one” data set.

```
data Demog;
infile  datalines truncover
firstobs=3;
input @1 name $char5. @7 ID 1.
@10 Age 2. @13 weight 3.;
datalines;
          1          2
12345678901234567890
Russ  1  23 170
Yatzi 2  28 101
Mike  3  29 240
;
run;
```

```
Data Visits;
infile  datalines truncover firstobs=3;
input @1 ID 1. @5 Visit 2. @10 Systolic 3.;
datalines;
          1          2
12345678901234567890
1    1    120
1    2    110
1    3    105
2    1    100
2    2    100
3    1    140
;
```

```
data OOps;
  merge Demog (SortedBy= ID)
        visits(SortedBy= ID);
  by ID;
  label Age = "Age in Months";
  Age = Age*12;
run;

proc print data=oops;
run;
```

Results of the merge

Obs	name	ID	age	weight	Visit	Systolic
1	Russ	1	804	170	1	120
2	Russ	1	9648	170	2	110
3	Russ	1	115776	170	3	105
4	Yatzi	2	336	101	1	100
5	Yatzi	2	4032	101	2	100
6	Mike	3	780	240	1	140

FIGURE 27

Figure 27 has four panels and illustrates a merge with a retain problem. One panel shows a data set called Demog. Another panel shows a data set called Visits. The two are merged to create a data set called OOps. The final panel has the results of the merge. The variable age is the problem. Our goal is to convert age in years to age in months and store the new value in the original variable. Reusing a variable name is ALWAYS a bad idea. In the figures below we will use the data step debugger to show how this logical error occurred.

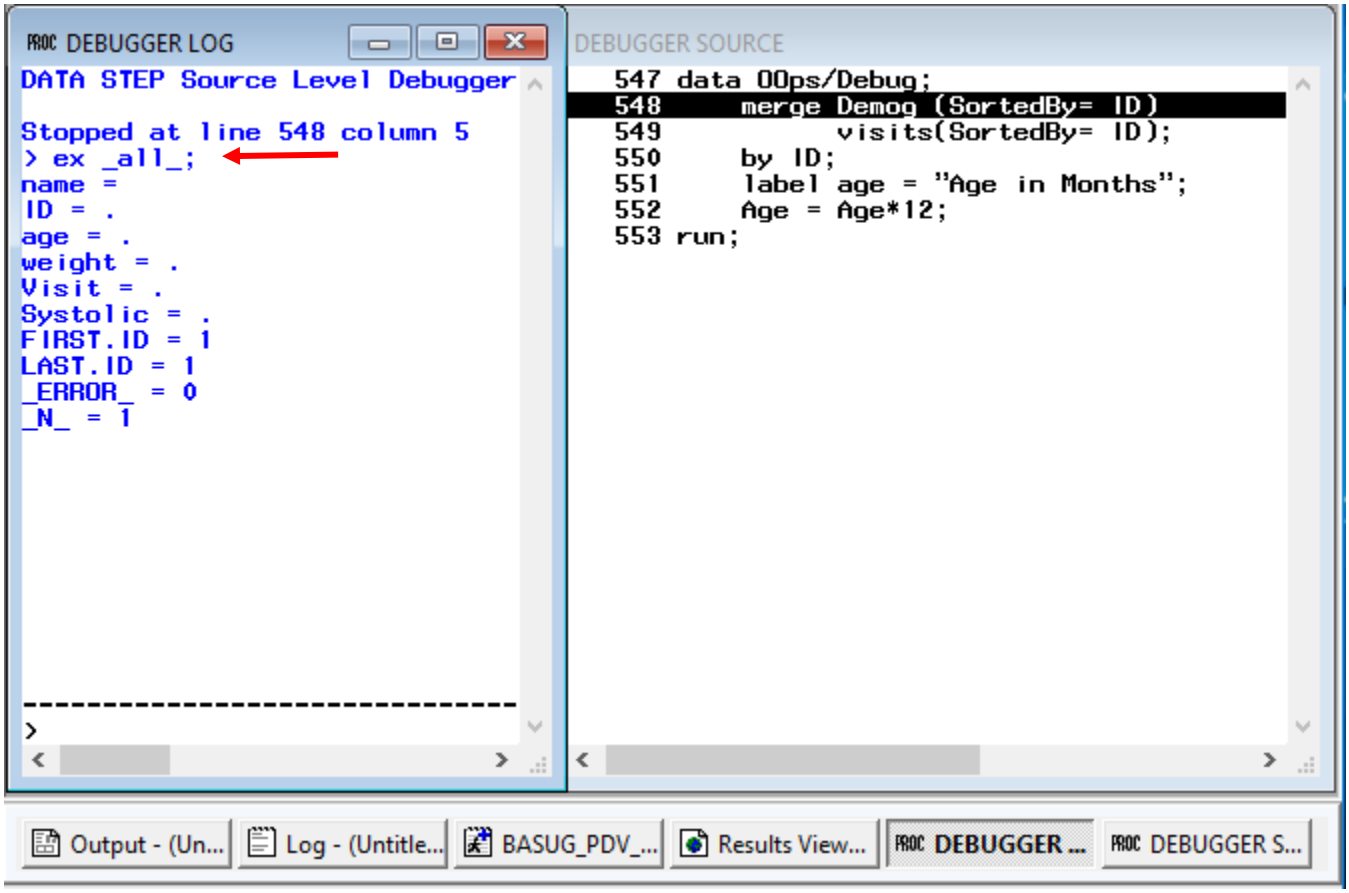

FIGURE 28

This is the first step of processing. The merge is about to execute and it data will appear in the next figure.

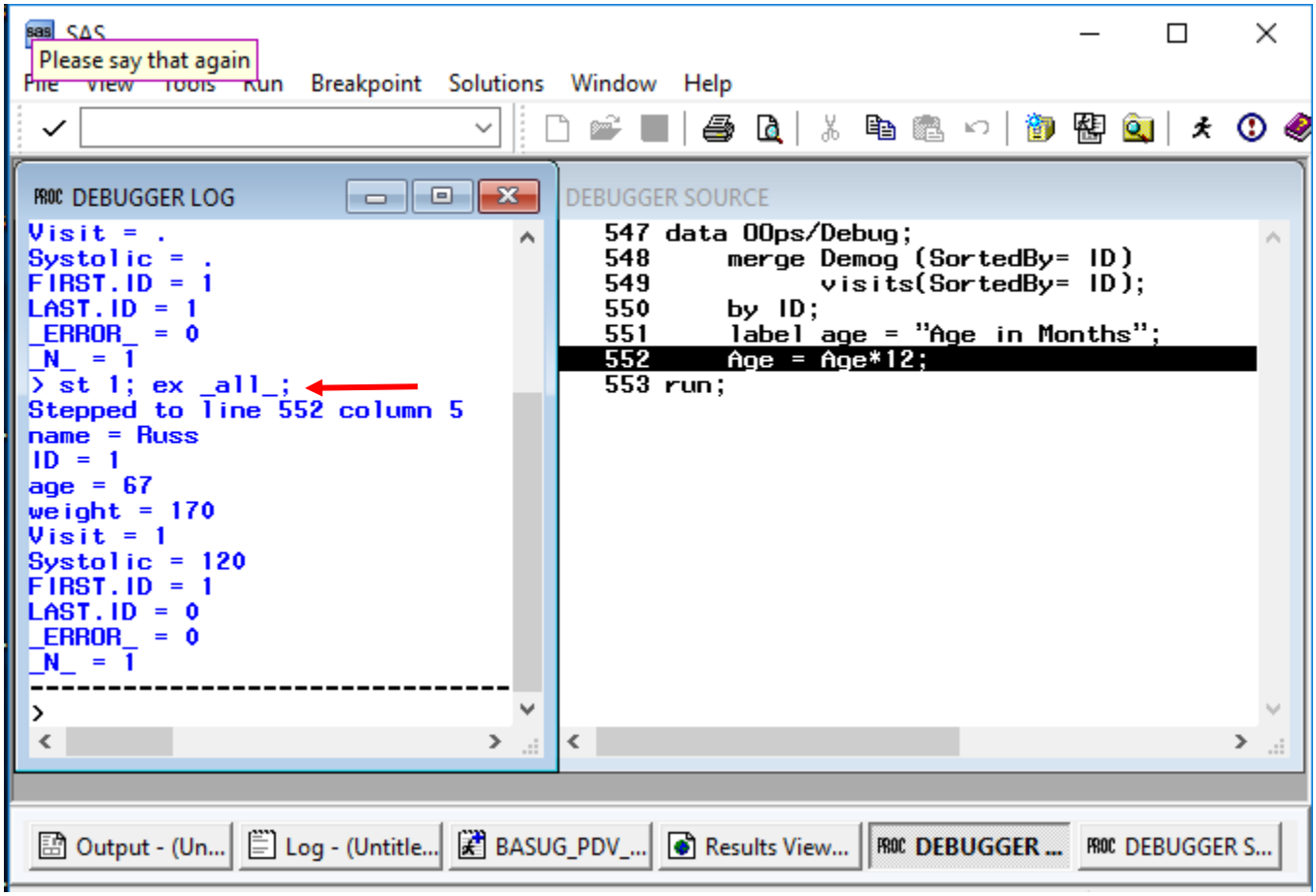

FIGURE 29

Line 548 just executed and we are about to execute line 552. Lines 548 to 551 are all compiler instructions and the debugger will jump the black high-lighted line over those lines. Note that the age, of 67, is the correct age in years. The problem is we're going to multiply age by 12 and then store the result back in age.

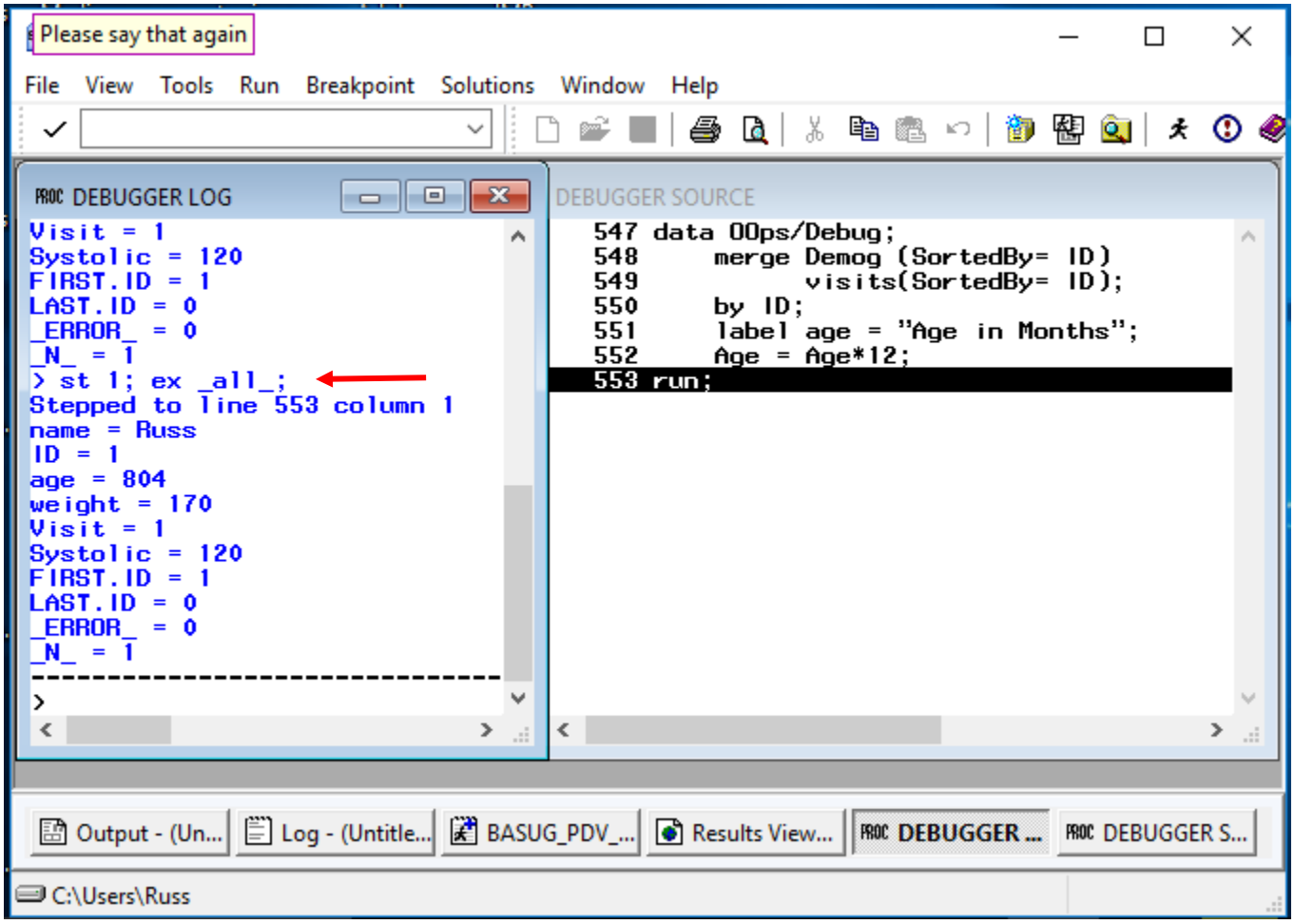

FIGURE 30

Line 552 just executed and we are about to execute line 553. Age now contains the correct number of months that this subject has been alive. The problem is that this value will be retained because, age originally came from the Demog data set and, therefore, the is automatically retained.

The next execution of the merge will read data – but only from one of the two input files. Because this is a one to many merge, it will only read data from the visits file.

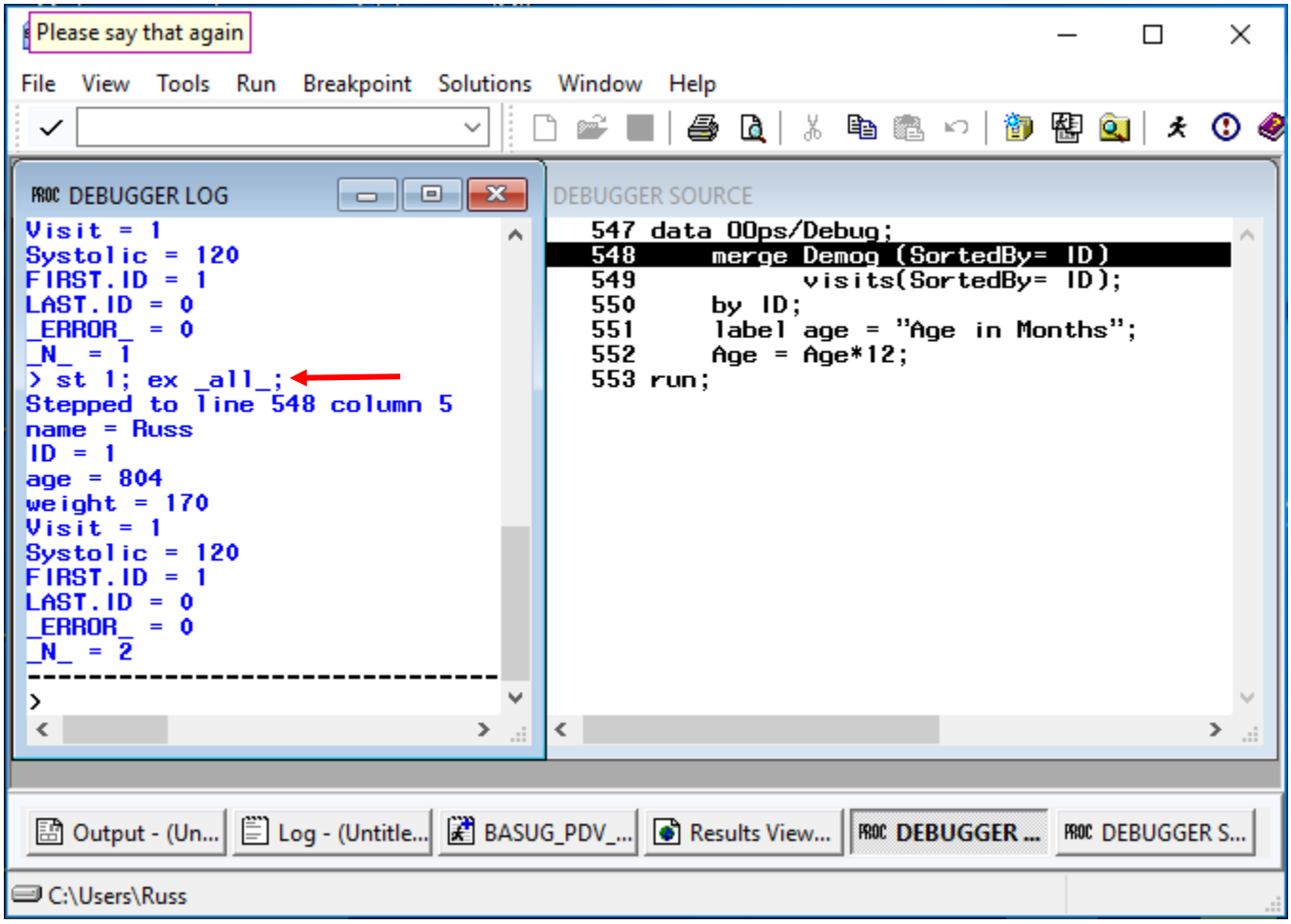

FIGURE 31

Line 553 just executed and control has passed through the top of the data step and down to line 548. We are about to execute line 548. As was said before, the data step is a loop and it reads a new observation (usually) when it makes another cycle through the loop. Notice that age has the value of 804 and that visit has the value of 1. The PDV, at this point, contains information from the previous “data read”.

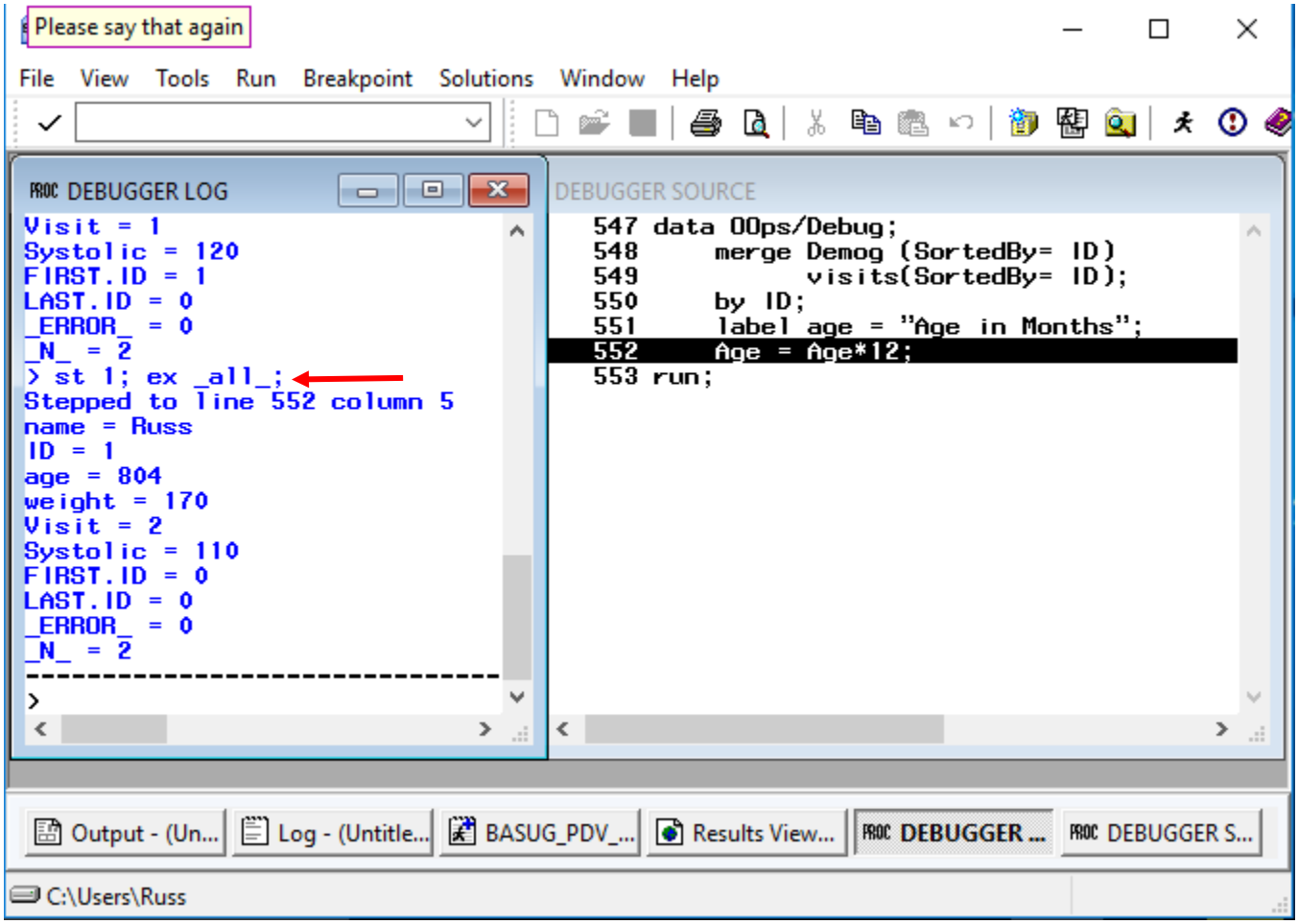

FIGURE 32

Line 548 just executed and we are about to execute line 552. As was said before, the data step is a loop and it reads a new observation (usually) when it makes another cycle through the loop. Line 548 read data – but only from the visits data set. Variables from Demog were retained and not changed. Notice that visit has the value of 2 and age still has the value of 804.

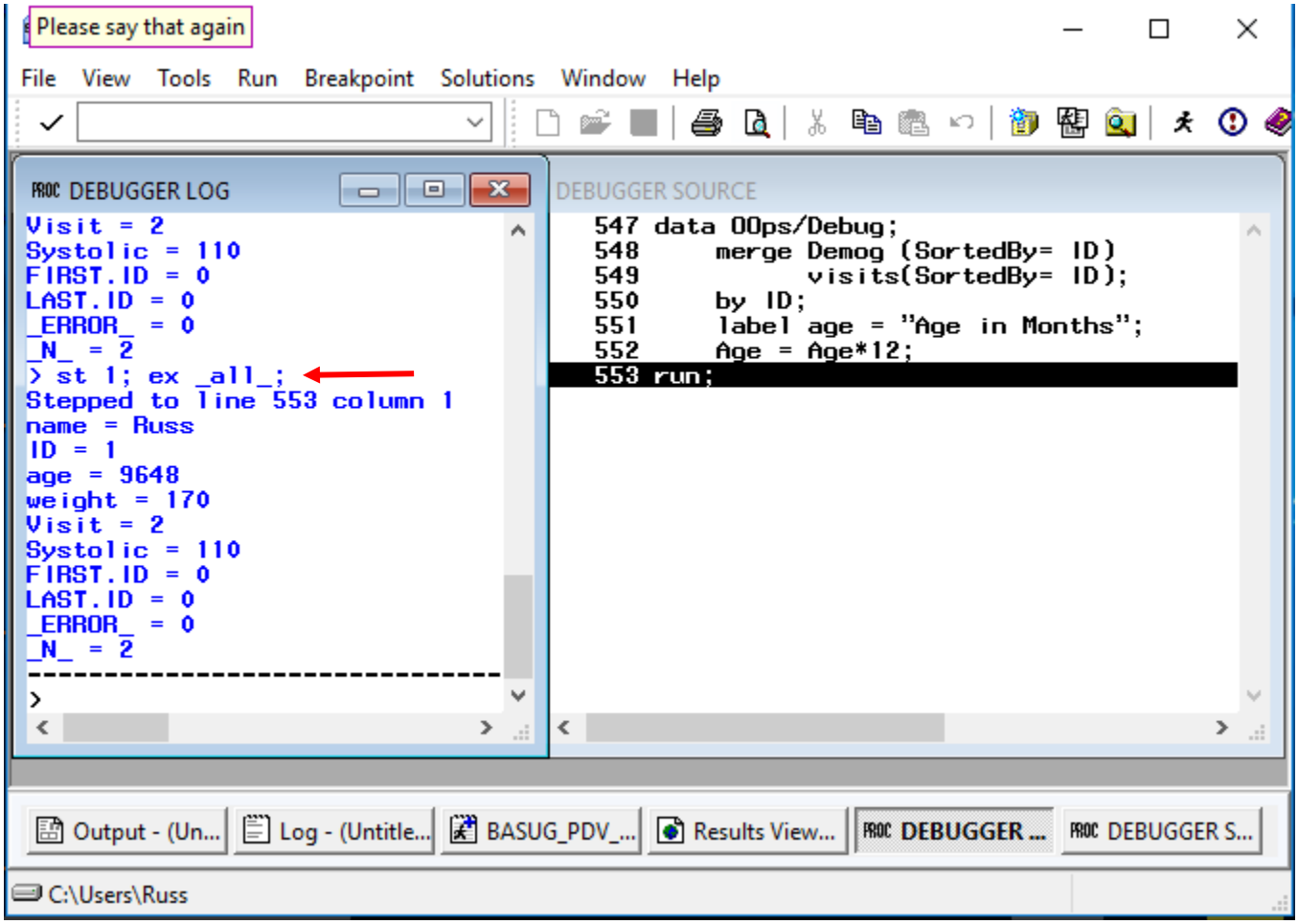

FIGURE 33

Line 552 just executed and we are about to execute line 553. Notice that visit has the value of 2 and age has the value of 9648.

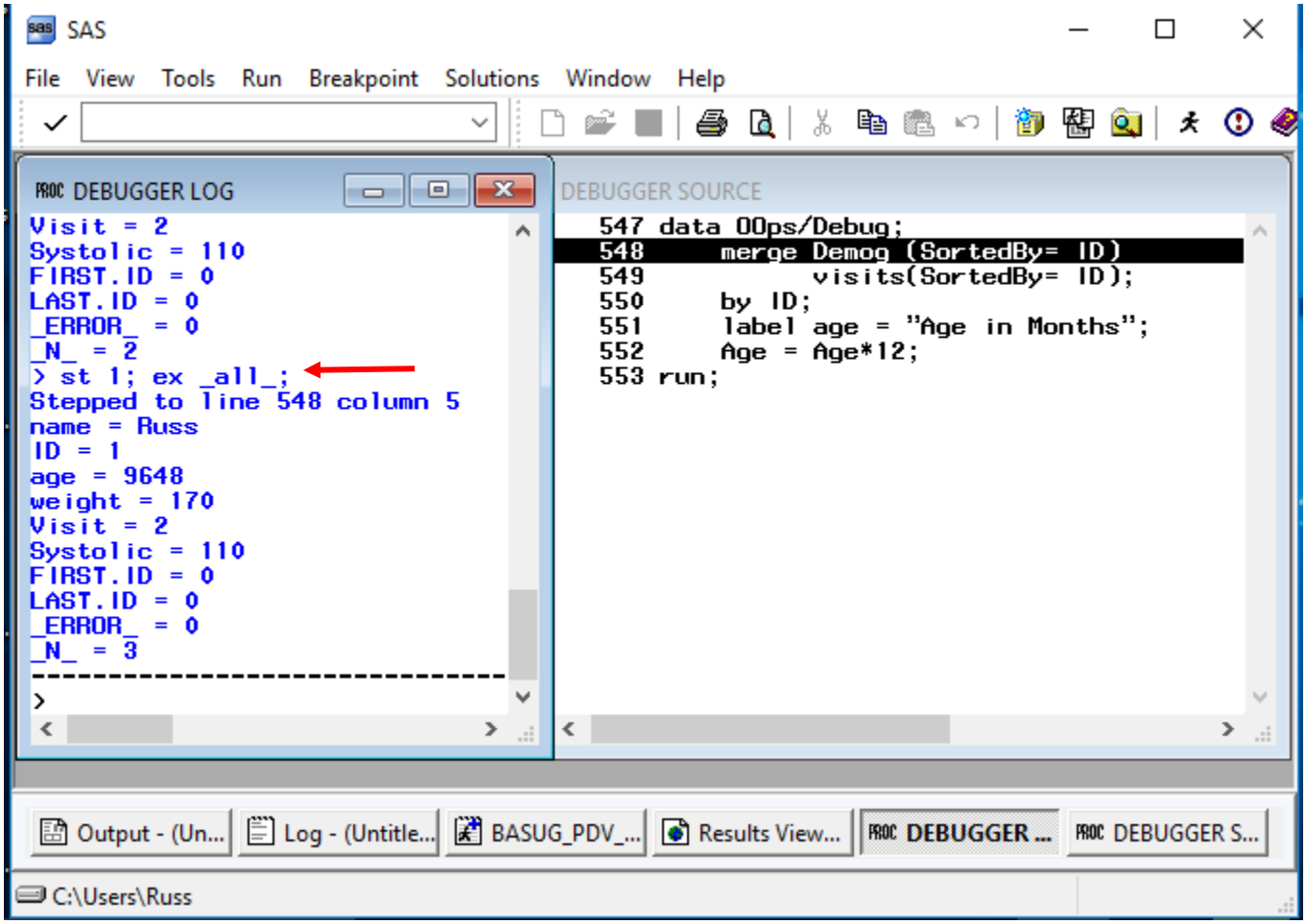

FIGURE 34

Line 553 just executed and control looped to the top (line 547) of the data step and back down to 548. We are about to execute line 548. That line will merge the one line of data for Russ (from the Demog table) with the third line of data for Russ (from the visits table). Note that the value for age is not correct. Please see values of the variable age in figure 27.

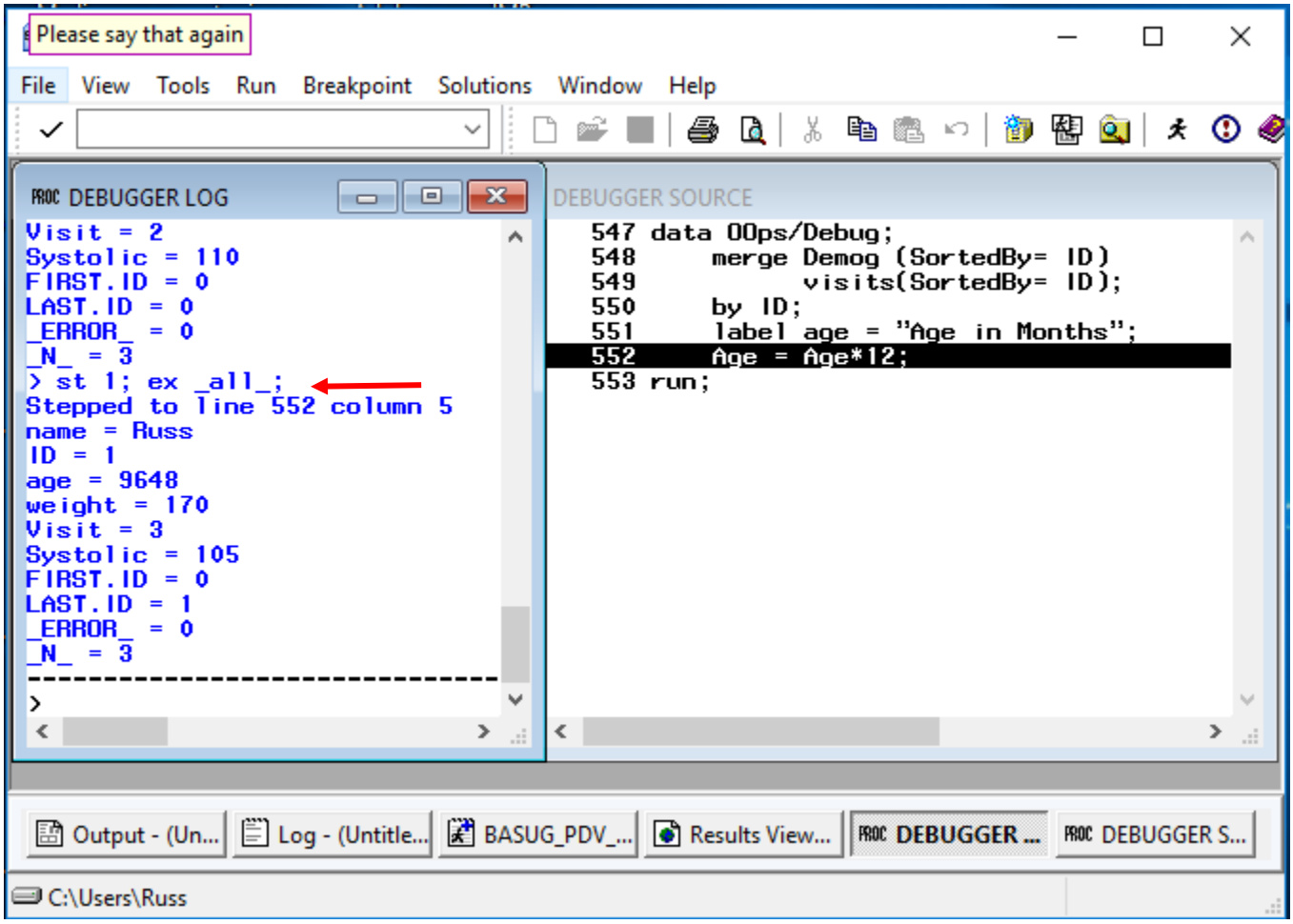

FIGURE 35

Line 548 just executed and we are about to execute line 552. The 9648 is from the last “loop”. Notice that visit has the value of 3 and we are about to multiply age by 12 another time.

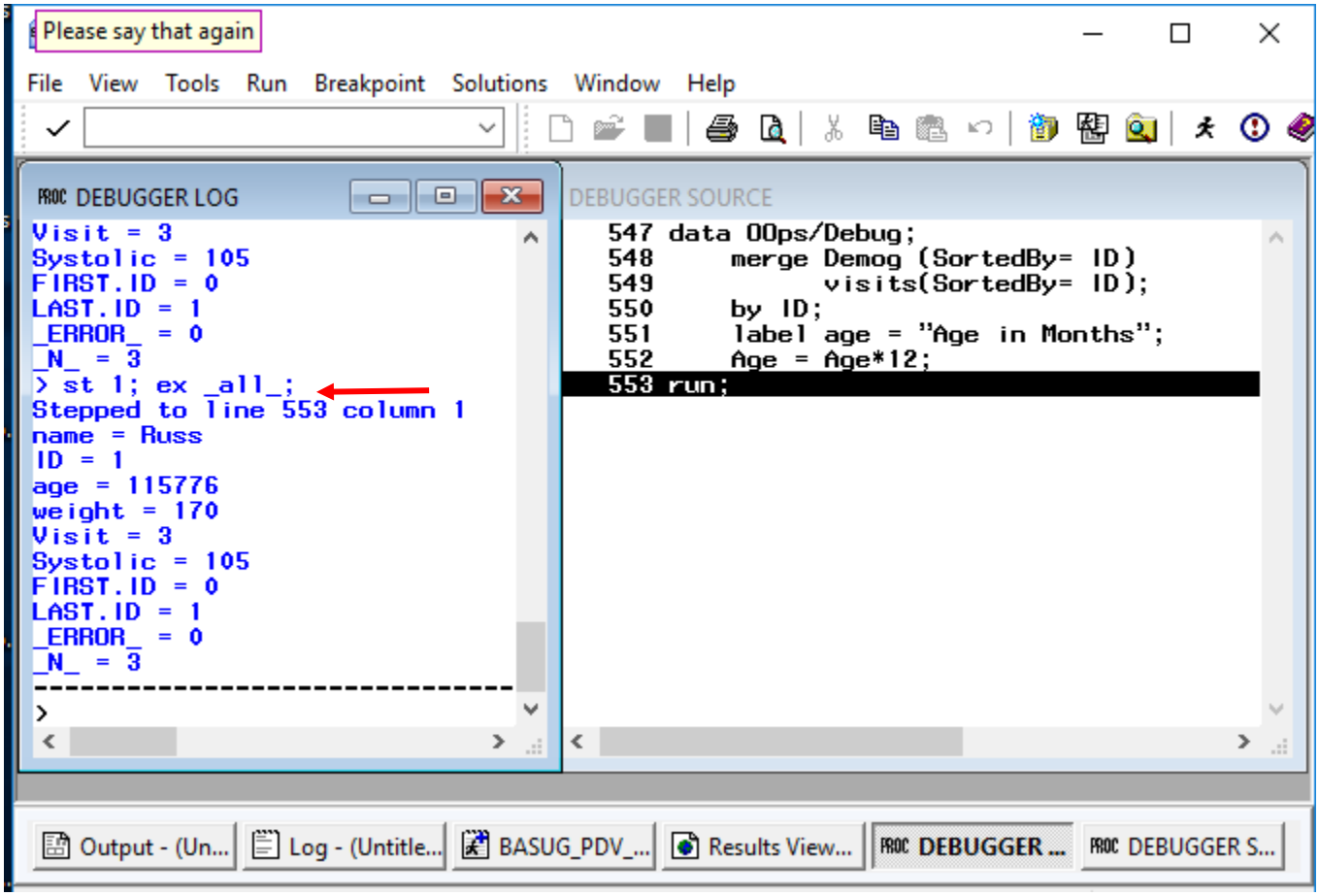

FIGURE 36

Line 552 just executed and we are about to execute line 553 (the run statement). Notice that visit has the value of 3 and age has the value of 115,776. The paper will not develop this issue any more. It should be noted that other subjects will have same problem. Note that it there is an easy fix for this problem and it is shown in the Figure below. The solution is to ***not overwrite the variable age*** when you are converting age to months. Re-using a variable is poor programming practice - always.

One should never "change the meaning of"/"re-use" a variable name or a data set name. Programs become much more difficult to debug if, in part of your program age means "age in years" and in another part of your program age means "age in months". The solution below is good programming practice.

```
data OOps2;
      merge Demog (SortedBy= ID)
            visits(SortedBy= ID);
      by ID;
      label age = "Age in Months";
      Age_mo = Age*12;
run;

proc print data=oops2;
run;
```

Obs	name	ID	age	weight	Visit	Systolic	Age_mo
1	Russ	1	67	170	1	120	804
2	Russ	1	67	170	2	110	804
3	Russ	1	67	170	3	105	804
4	Yatzi	2	28	101	1	100	336
5	Yatzi	2	28	101	2	100	336
6	Mike	3	65	240	1	140	780

FIGURE 37

EXAMPLE 3: A SUBSETTING WHERE VS A SUBSETTING IF

Example 3 illustrates the difference between the sub-setting where and the sub-setting if. The data set is sorted by sex and so the first several observations in the data set are all females. If one codes “where sex equals ‘M’” the first observation that comes into the program data vector is a male. If one codes ”if sex equals ‘M’” the first observation that comes into the program data vector is a female. Sub-setting where clauses run faster than sub-setting if clauses.

Sub-setting WHERE	Sub-setting IF

`Data Use_where /debug;`
`  set sorted_Class;`
`  where sex="M";`
 	
`  RUN;`
	`Data Use_if /debug;`
	`  set sorted_Class;`
	`  if sex="M";`
	
	`  RUN;`

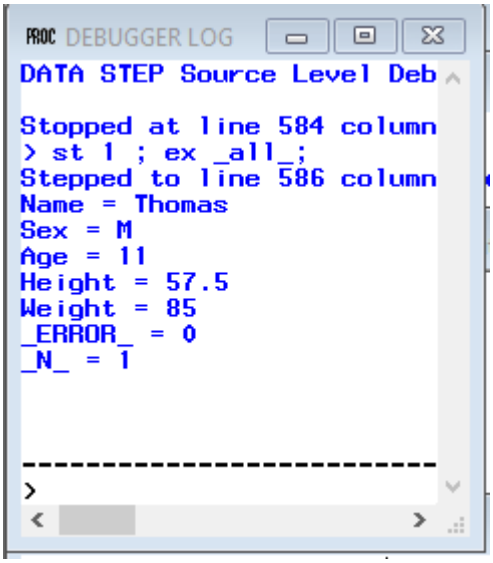	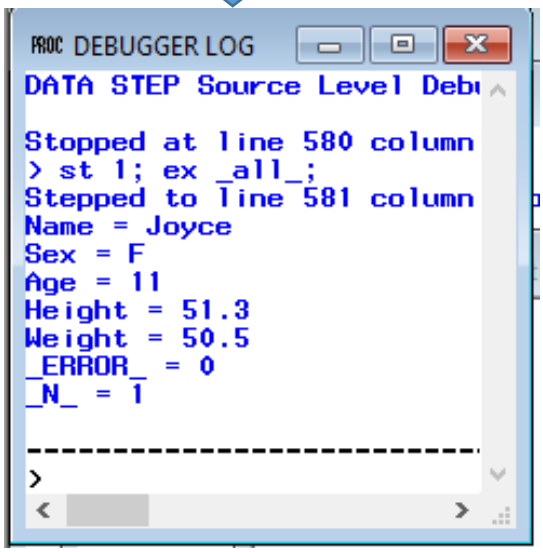
FIGURE 38	

EXAMPLE 4 THE DOW LOOP

It is hoped that the reader never finishes reading this example. It is hoped that the reader gets part way through the example and says “I can do this ***myself*** in the debugger”.

```
DATA Stu_Grades;
 infile datalines firstobs=3 trun-
cover;
INPUT @1 name $char5. @7 sex $char1.
      @9 exam 1. @12 grade 3.;
DATALINES;
          1          2
12345678901234567890
Aruna f 1  90
Aruna f 2  93
Zihui f 3  97
Zihui f 1  95
Zihui f 2  95
;
PROC SORT DATA=Stu_Grades;
BY name;
run;
```

```
DATA Stu_Averages;
 Tests_taken = 0;
 Grade_total = 0;
DO UNTIL (LAST.name);
    SET Stu_Grades;
    BY name;
    Tests_taken + 1;
    Grade_total + grade;
END;
Avg_Grade = Grade_total /
Tests_taken;
run;

 PROC PRINT DATA=STU_AVERAGES; RUN;
```

```
PROC PRINT DATA=STU_GRADES; RUN;
```

FIGURE 39

A characteristic of the DOW loop and not the implied loop the controls the data step. When using the DOW loop, it is very common to read a data set with thousands of rows and have, at the end of processing, _N_ have a value of only one or two.

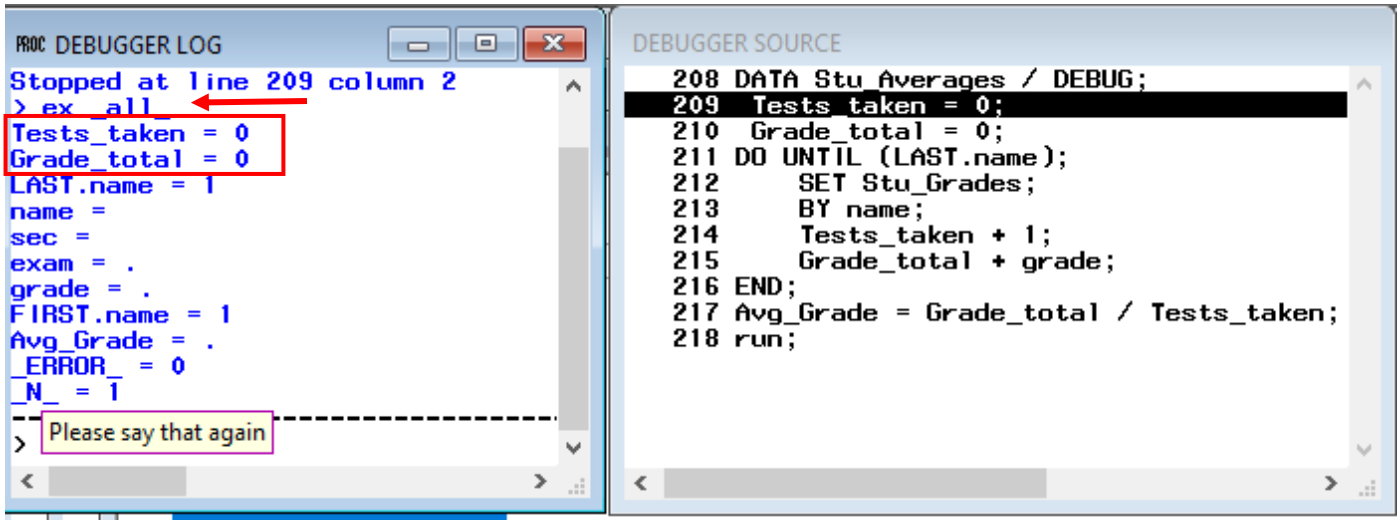

FIGURE 40

line 208 was processed and line 209 is about to execute. This figure shows the condition of the PDV before execution. Tests_taken and Grade total are valued with zeros because the incrementing syntax was of the form x+1. First.name and Last.name are valued as 1, because we are processing the one row where name is blank

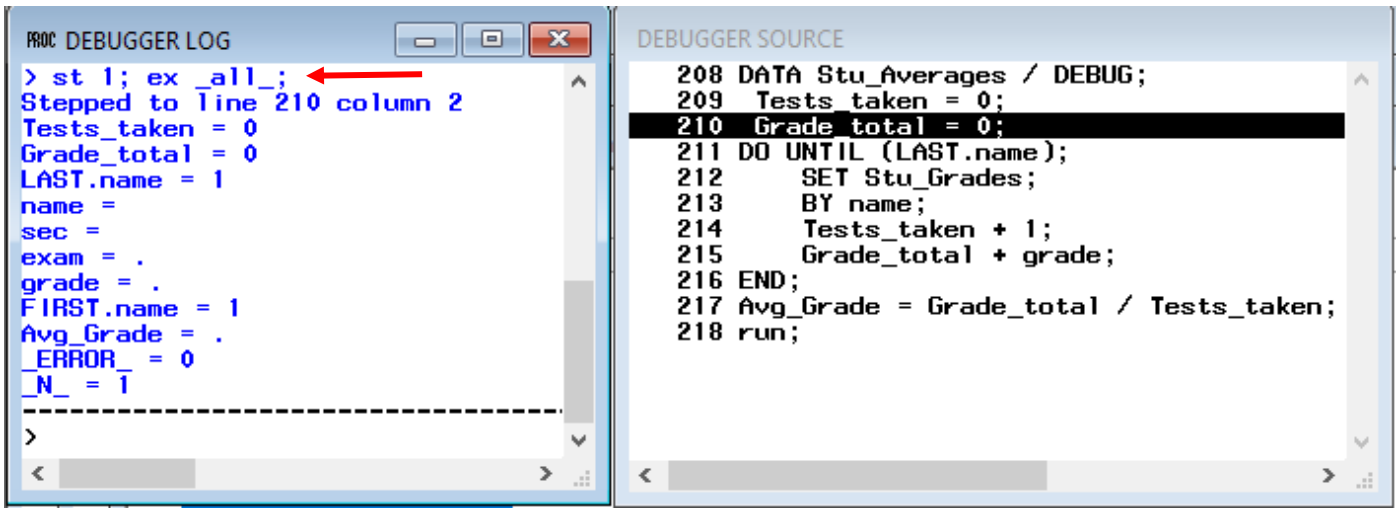

FIGURE 41

Line 209 has just executed and line 210 is about to execute. Tests_taken was just set to zero. First.name and Last.name are valued at 1 because this “row” is the only row (therfore the first and last row) that has a missing name We will not read data until line 212 processes,

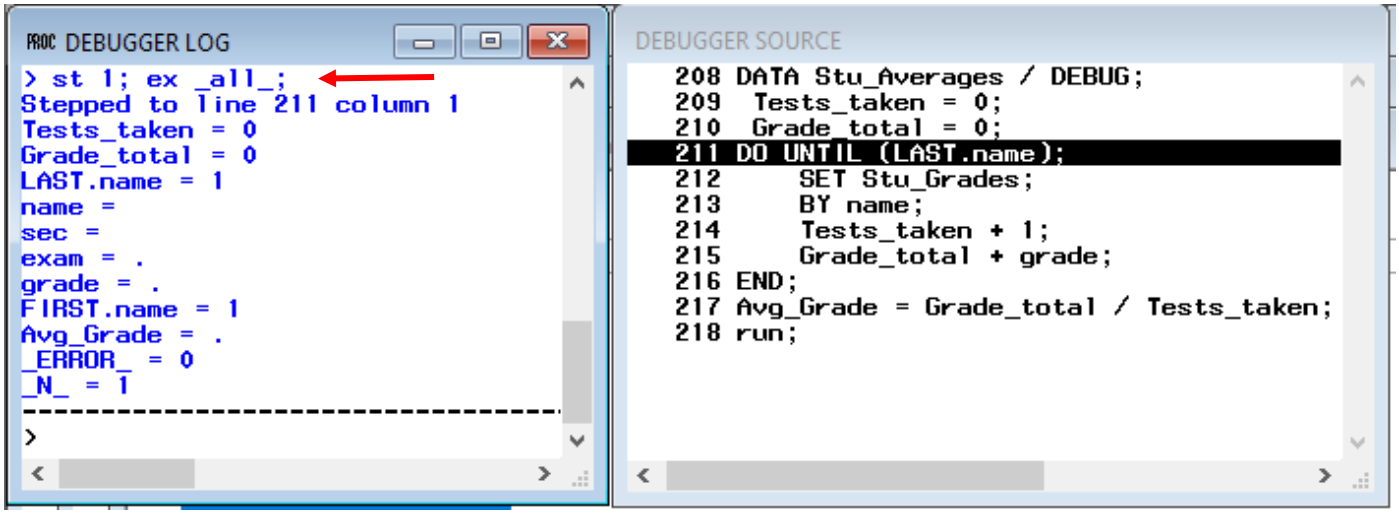

FIGURE 42
Line 210 has just executed and line 211 is about to execute.

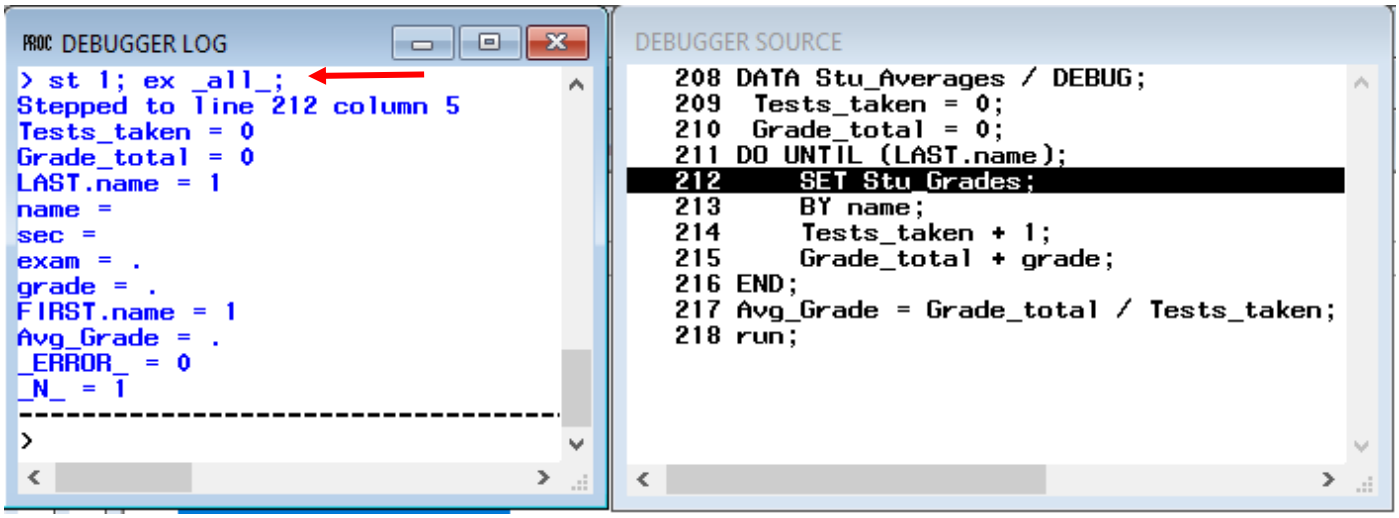

FIGURE 43
Line 211 has just executed and line 212 is about to execute.

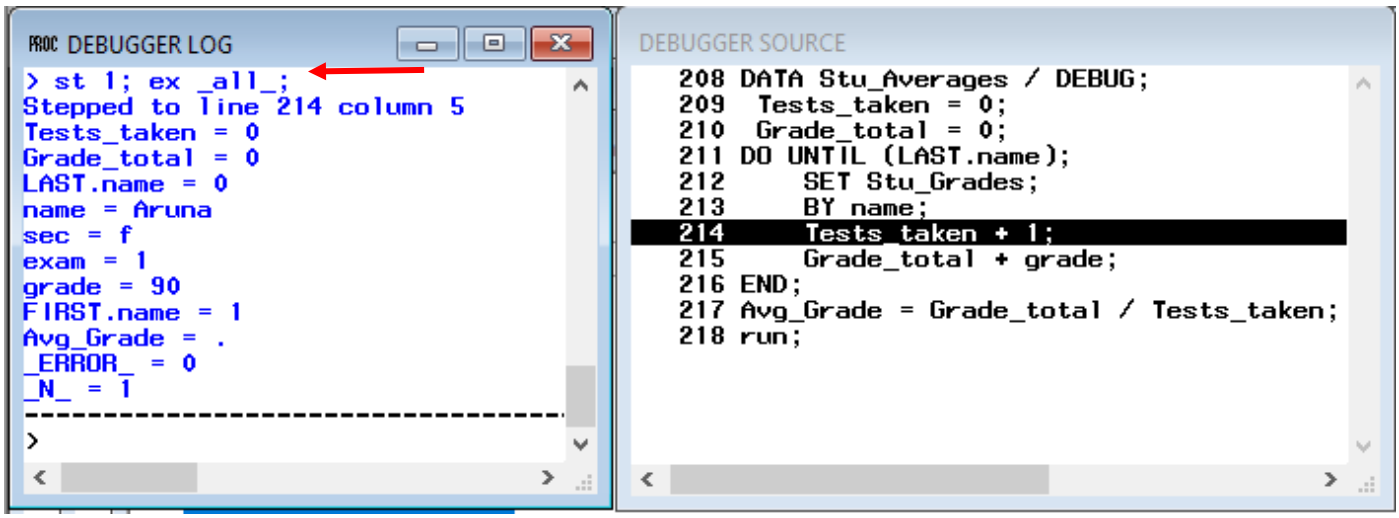

FIGURE 44
Line 212 has just executed and line 214 is about to execute. Data was read into the PDV.

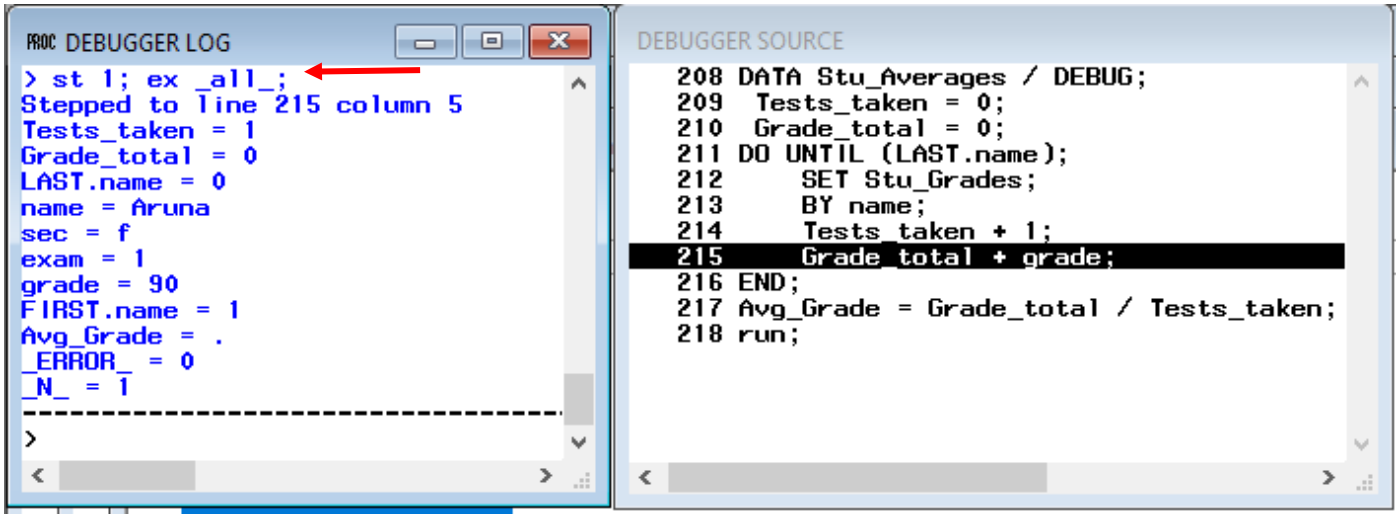

FIGURE 45
Line 214 has just executed and line 215 is about to execute. Tests_taken has the value of 1

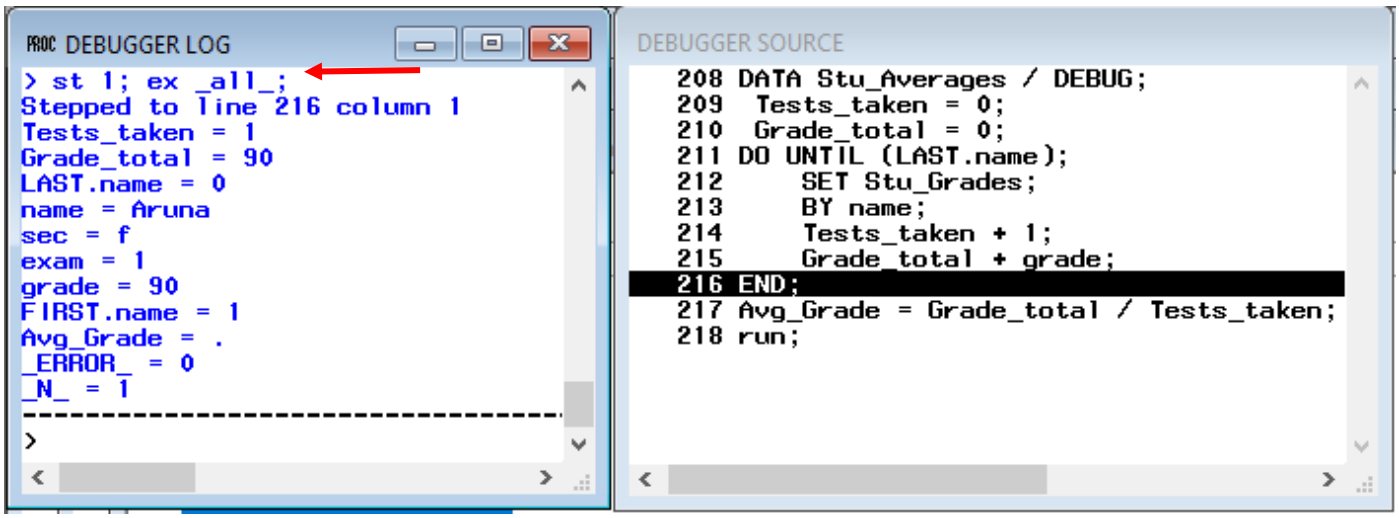

FIGURE 46
Line 215 has just executed and line 216 is about to execute. Tests_taken has the value of 1. Grade_total as the value of 90. Aruna took two tests so control will loop back to 211 and we will process her second test.

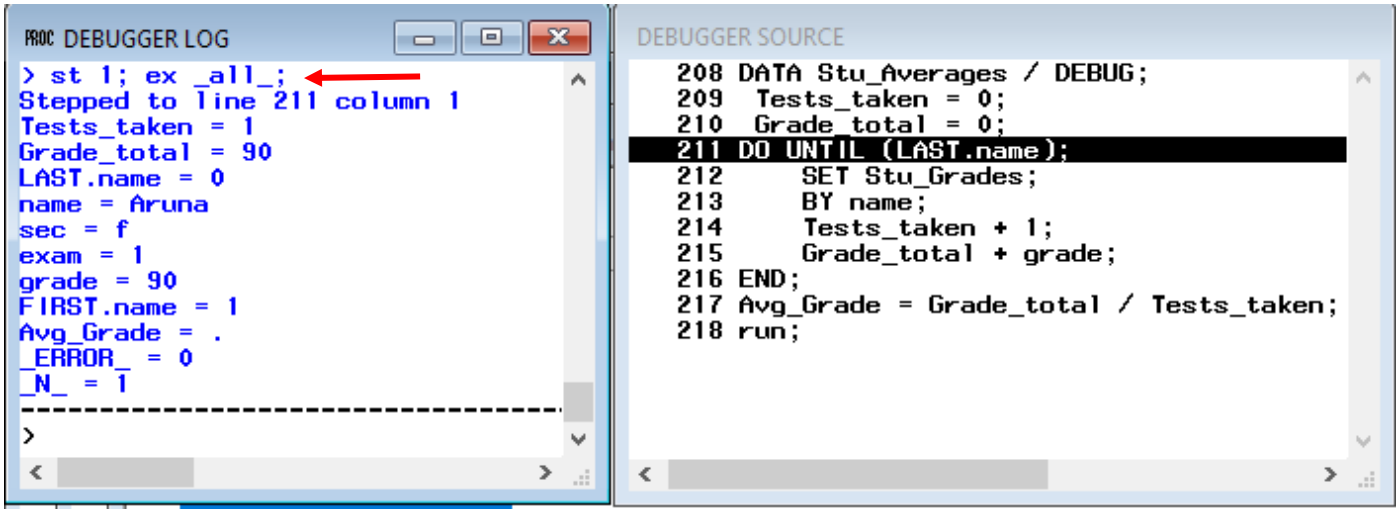

FIGURE 47
Line 216 has just executed (that was the last line of the loop) and 211 is about to execute. Control shifted back to line 211 – the top of the loop.

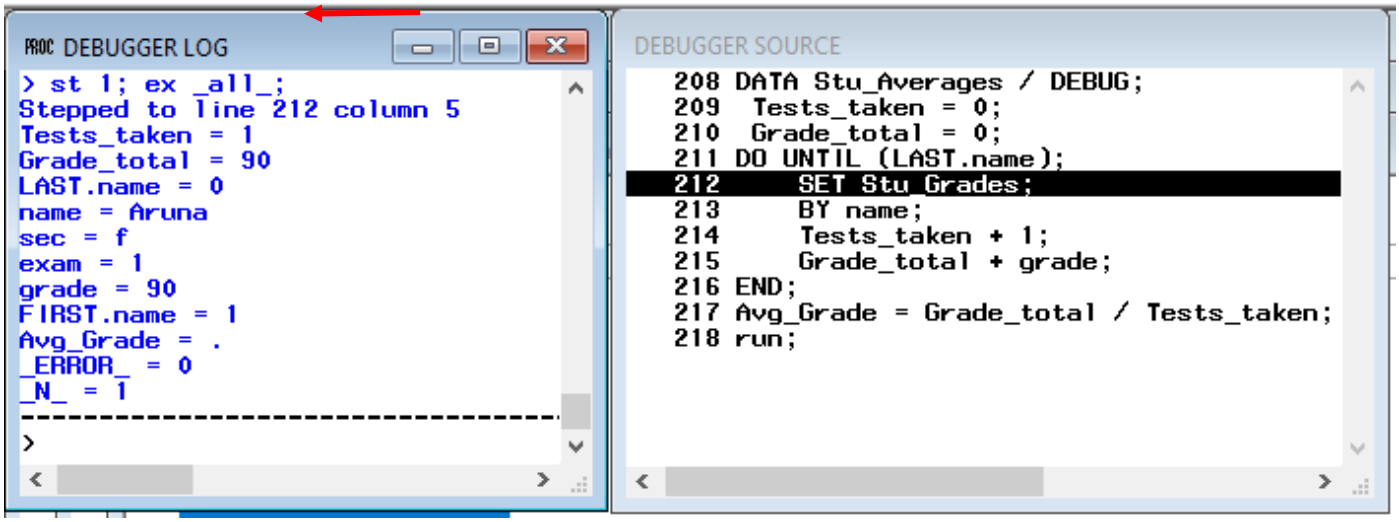

FIGURE 48
Line 211 has just executed and 212 is about to execute. We are about to read data.

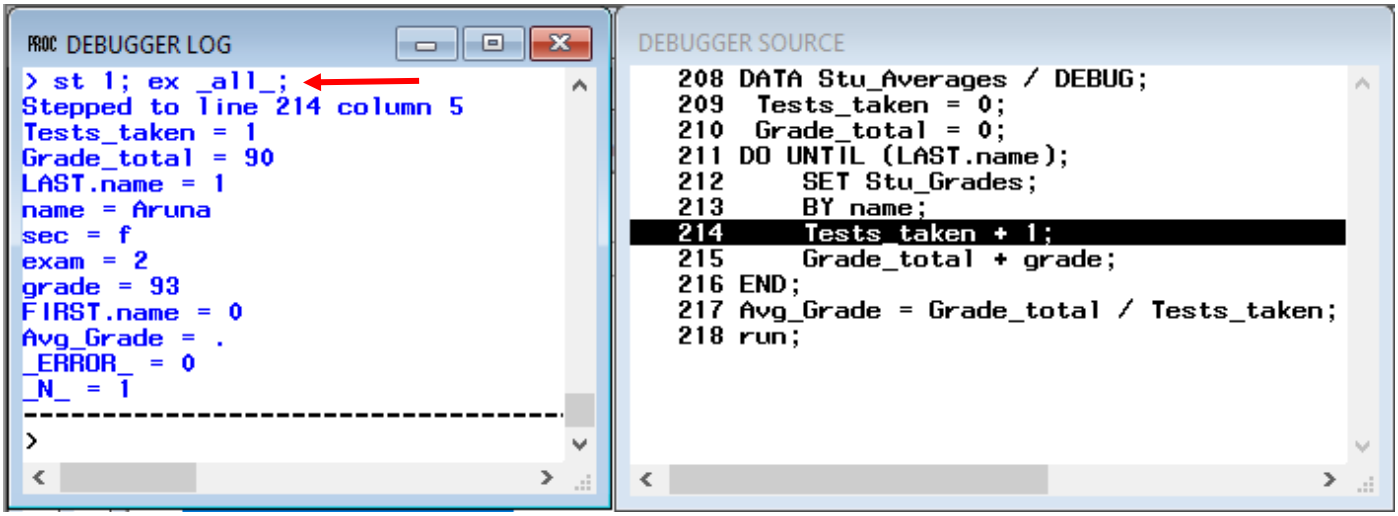

FIGURE 49
Line 212 has just executed and 214 is about to execute. New data, for Aruna's second exam, was read into the PDV.

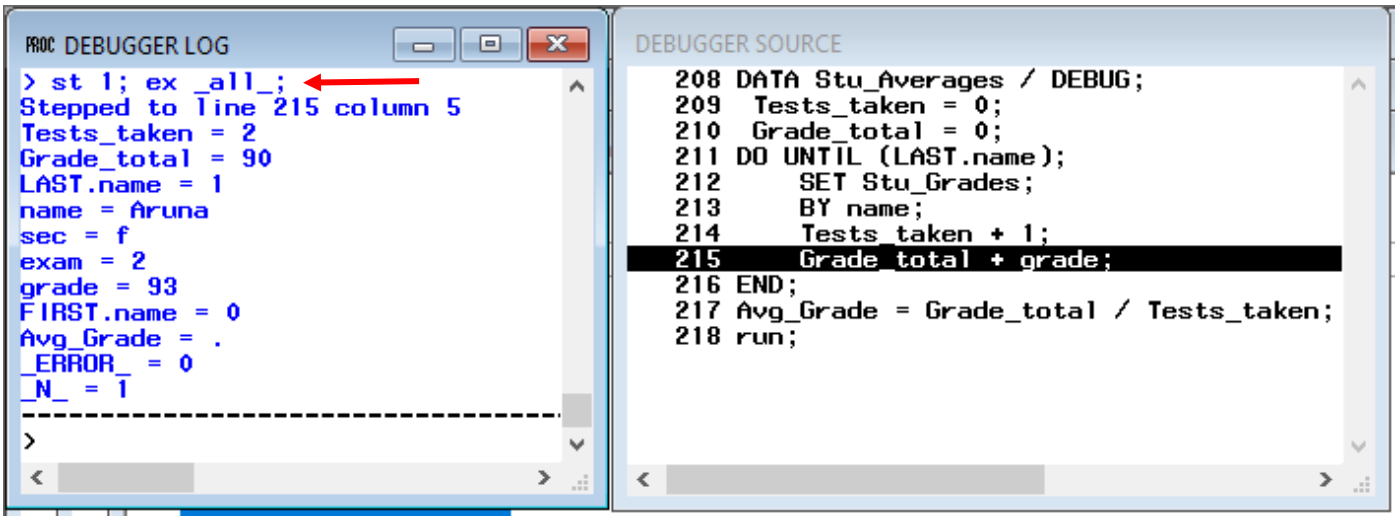

FIGURE 50
Line 214 has just executed and 215 is about to execute. Tests_taken has the value of 2;

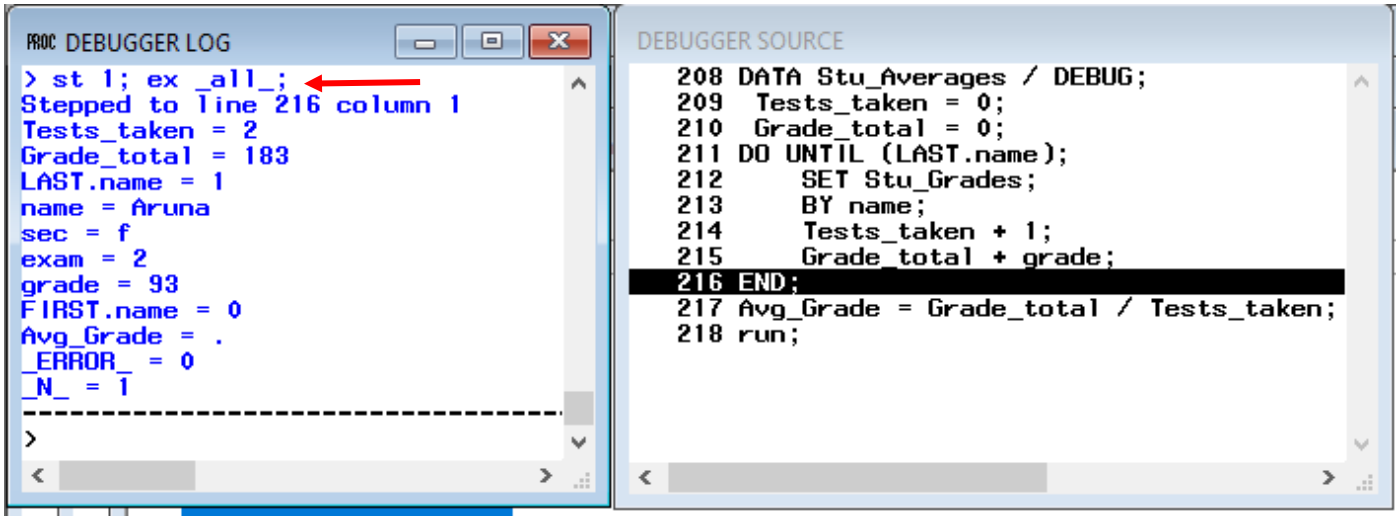

FIGURE 51
Line 215 has just executed and 216 is about to execute. Grade_total has the value of 183. Last.name has a value of one so control will escape the loop.

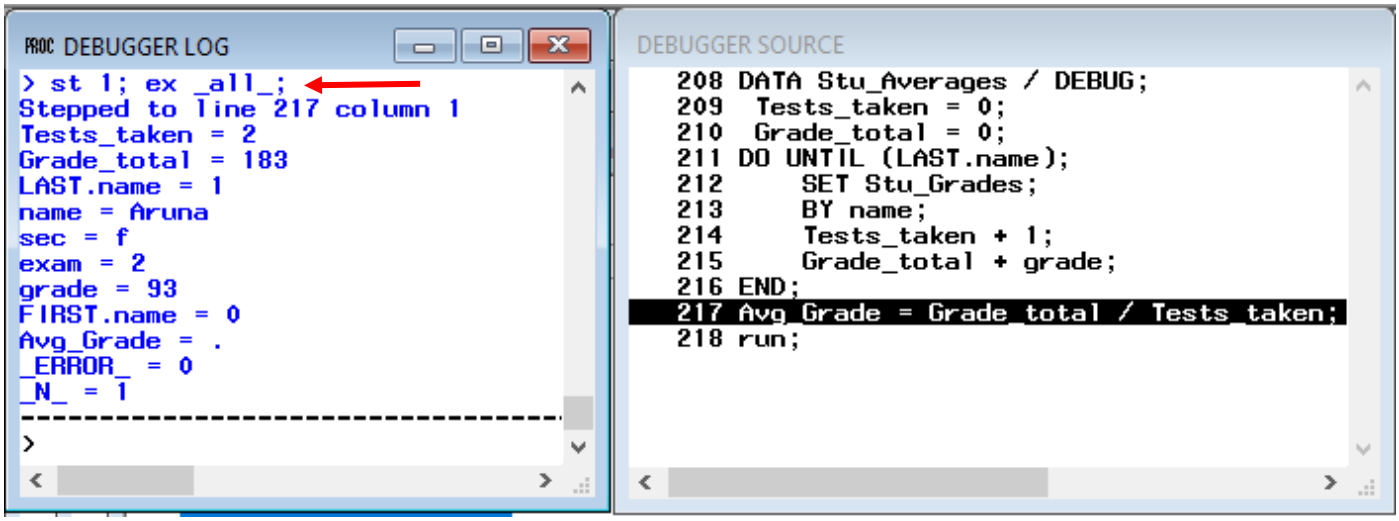

FIGURE 52
Line 216 executed. Control escape the loop and 217 is about to execute. Grade_total is valued at 183; "Do Until"l logic is checked at the "bottom" (216) of the loop and this is the last row of data for Aruna. Control exits the loop.

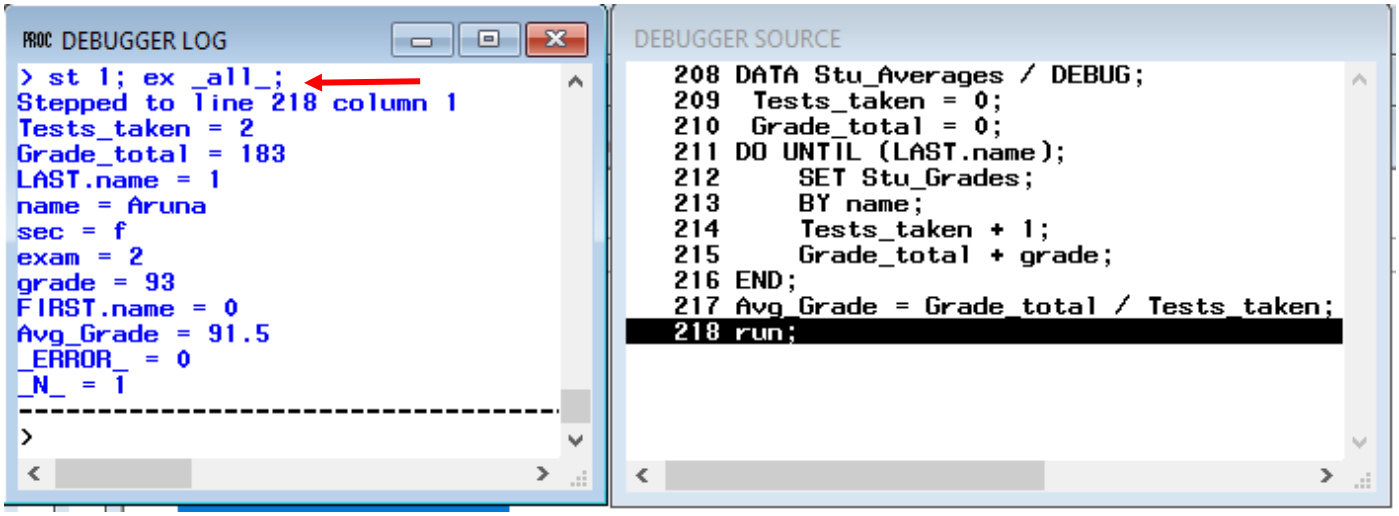

FIGURE 53
Line 217 has just executed and 218 is about to execute. The loop between lines 211 to 216 controlled processing for one student. We have now finished processing the data for student number one and will loop to the top of the data step to start processing the data for student number two.

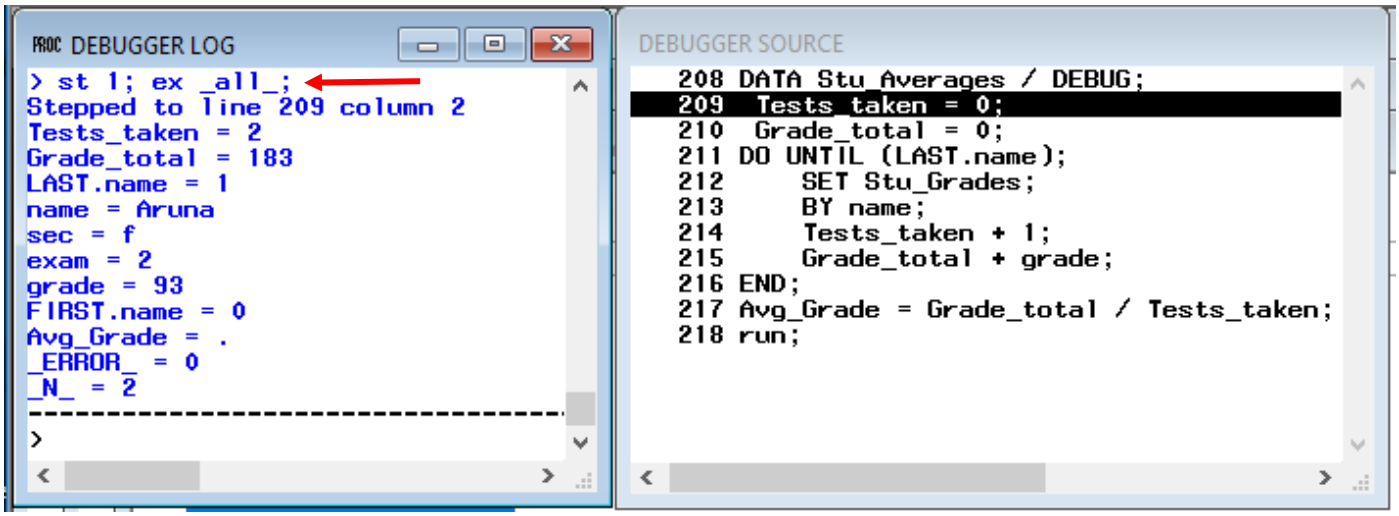

FIGURE 54
Line 218 executed and control looped to the top of the data step. Line 209 will execute next. When control passes through the top line of the data step,calculated variables, like Avg_grade, are set to missing. _N_ is incremented. First-dot and Last-dot Variables have not been changed and still contain values from the last observation processed.

Line 209 is about to execute. It will reinitialize tests_taken to zero in preparation for processing another student.

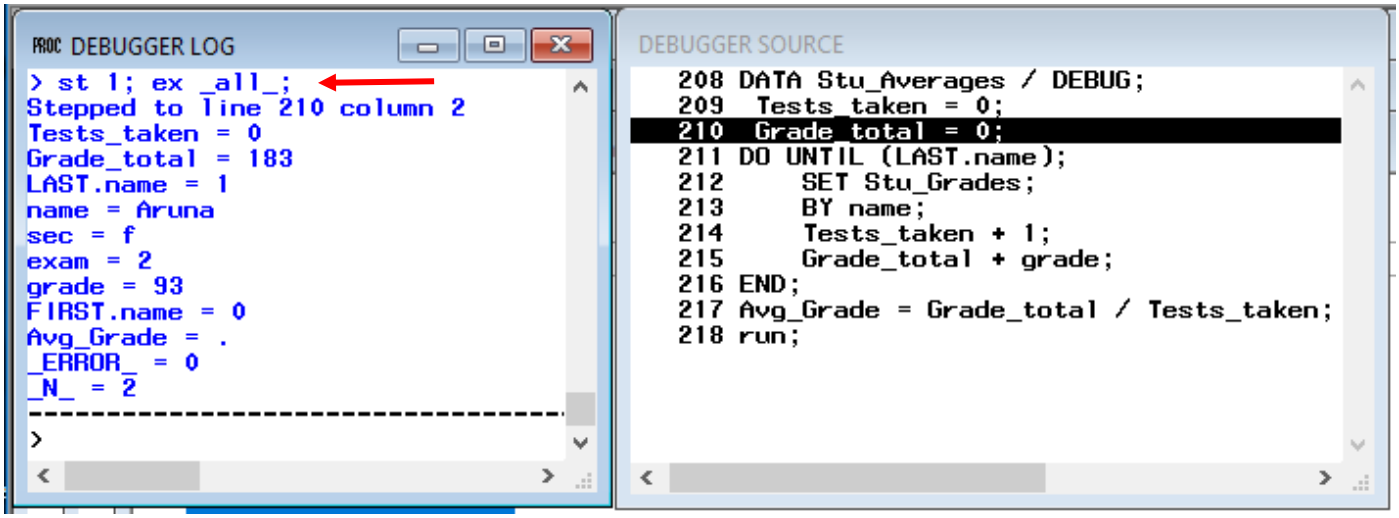

FIGURE 55
Line 209 executed and 210 is about to execute. Line 210 will reinitialize Grade_total to zero in preparation for processing another student.

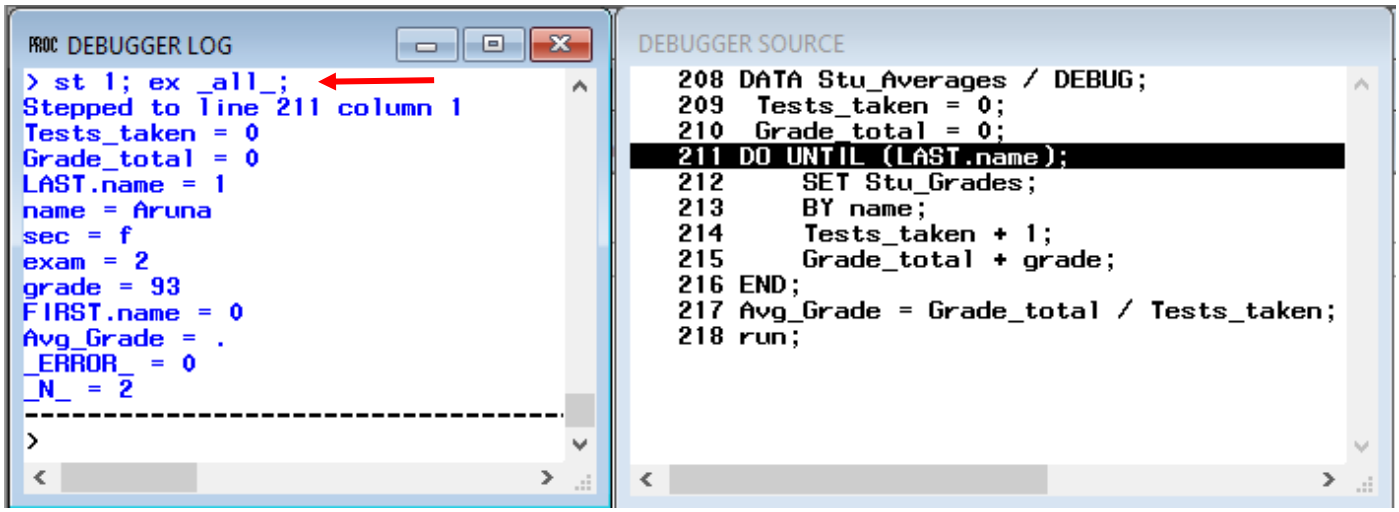

FIGURE 56
Line 210 executed and 211 is about to execute. Line 211 will start the looping process that reads data for another student.

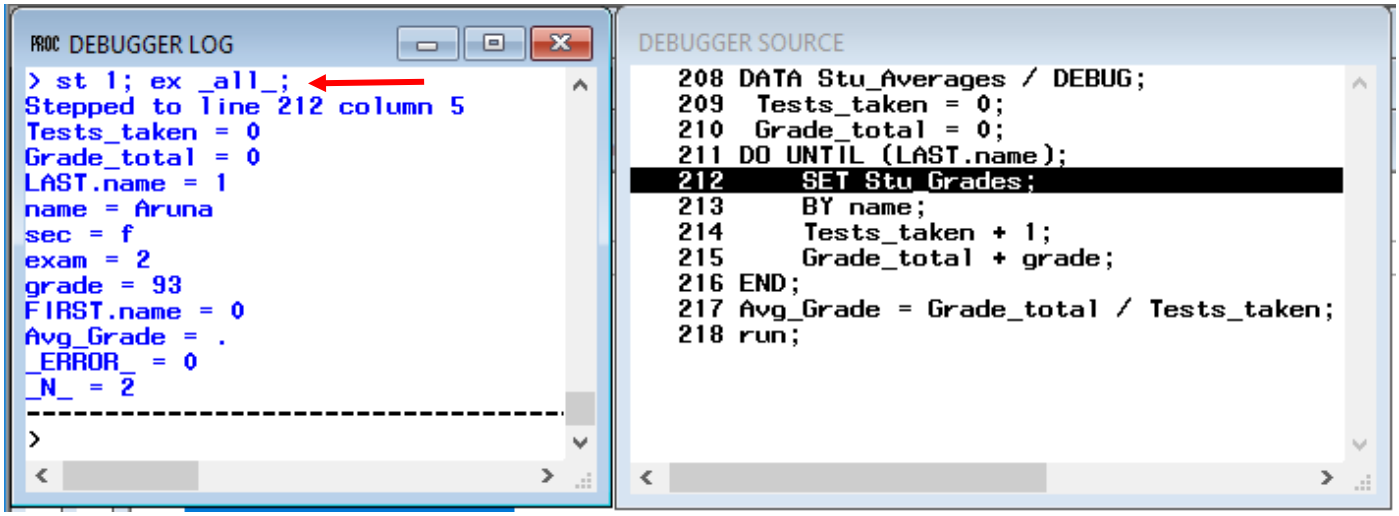

FIGURE 57
Line 211 executed and 212 is about to execute. We are about to read data.

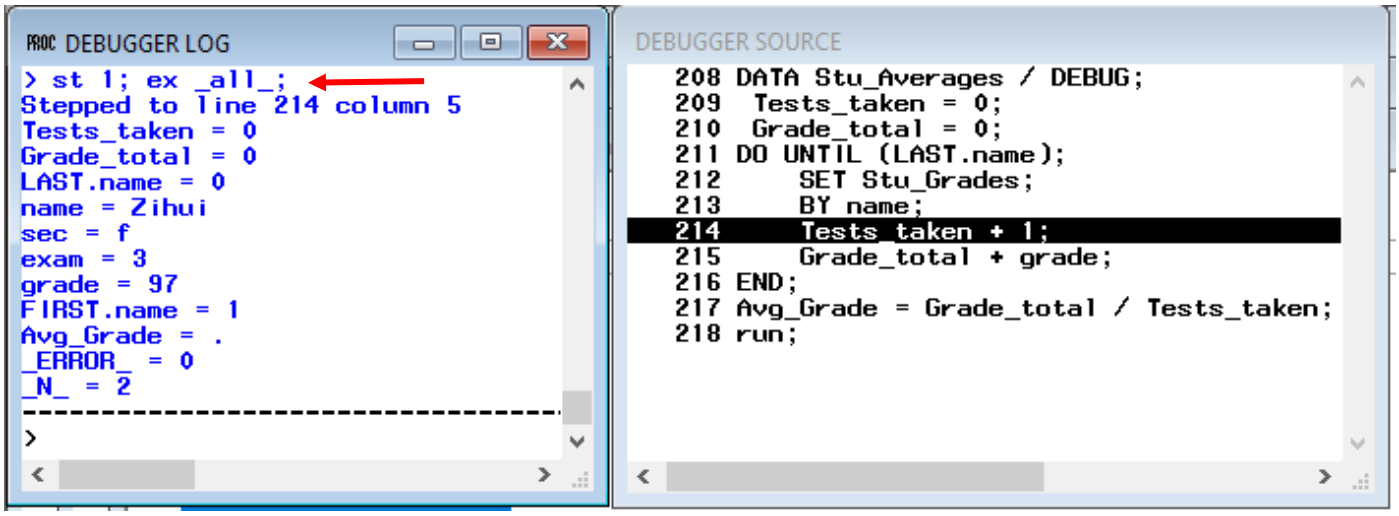

FIGURE 58
Line 212 executed and 214 is about to execute. Line 212 read data for a new student into the PDV. Note that the values of First.name and Last.name have changed from the values in Figure 57.

EXAMPLE 5 _TEMPORARY_ ARRAYS DO NOT NORMALLY SHOW

```
data show_temp /debug;
  array alone(4) _temporary_;
  Noote ="This is just to show temp array variables";
  set sashelp.class;
   do i=1 to 4;
     alone(i)=i*2;
   end;

  RUN;
```

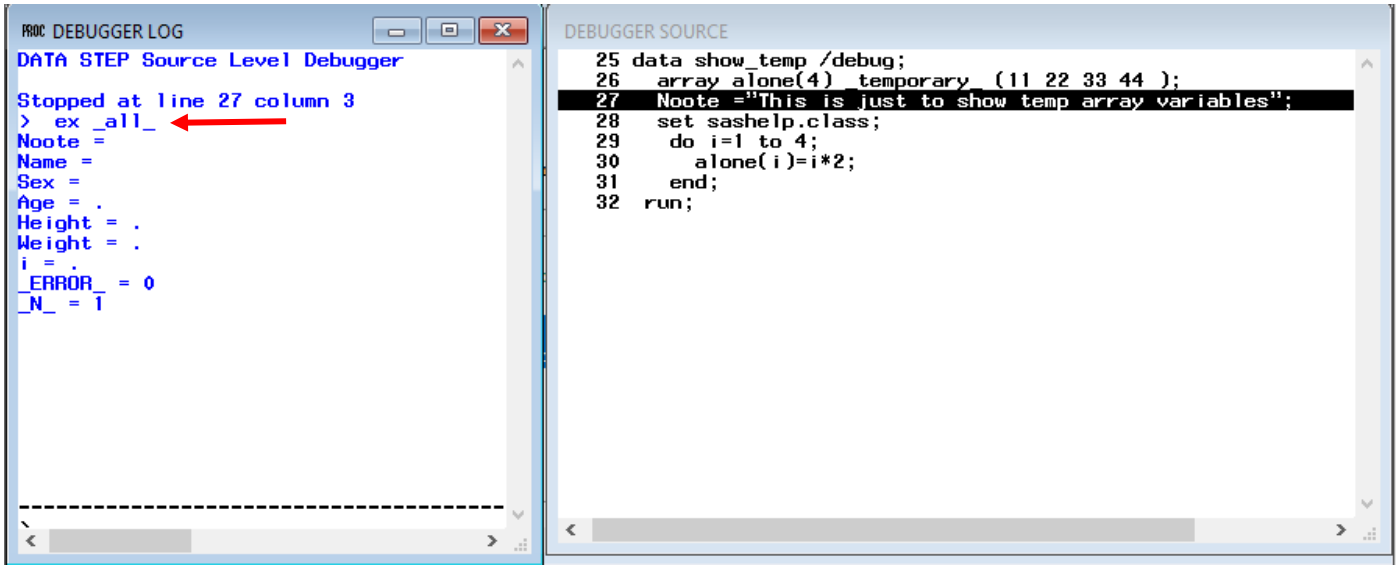

FIGURE 59
The code above and the screenshot of the data step debugger both show that we created an array using temporary variables. Figure 59 shows that the temporary variables do not appear to be on the program data vector when we issue the command ex _all_;

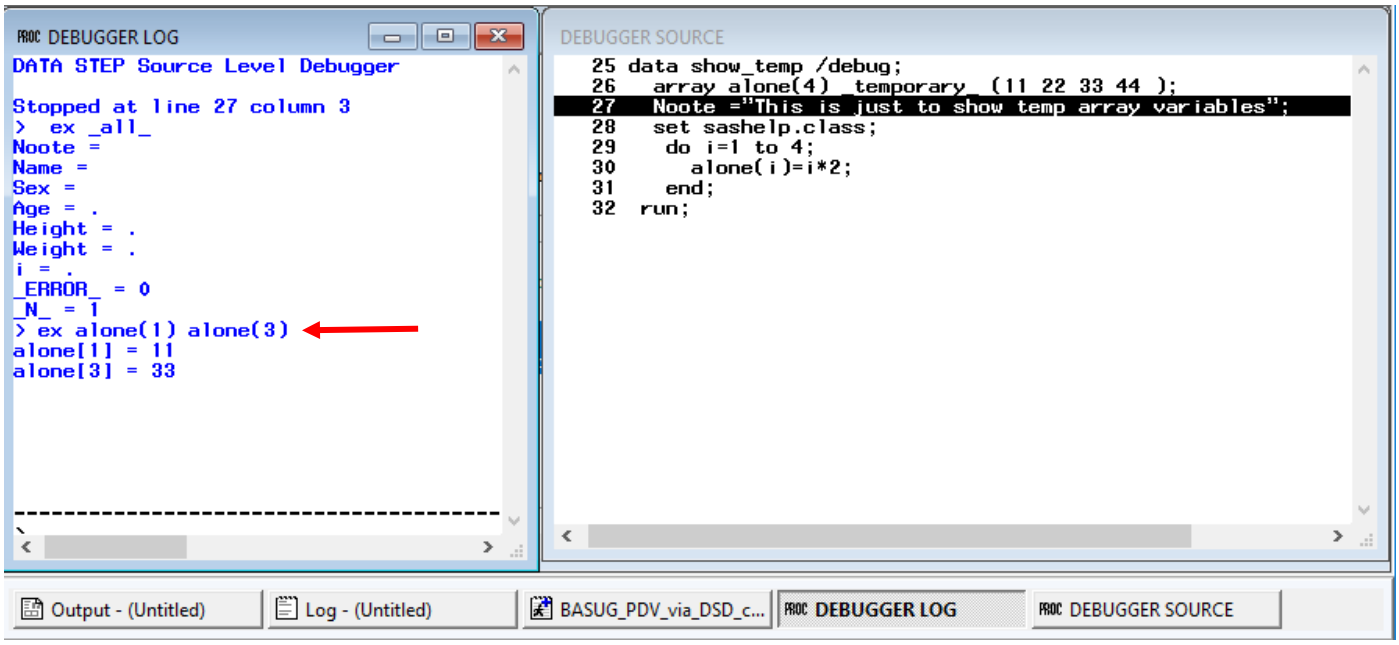

FIGURE 60
however; if we ask to examine the variables using their names they do show up in the debugger log .

SUMMARY

Understanding the program data vector is very important in debugging complicated data steps. The SAS data step debugger is an excellent tool to understand the program data vector and to help find out what might be going wrong in your complicated data step.

REFERENCES

Brucken, Nancy. PROC TRANSPOSEs = 1 DATA step DOW-Loop.

Available atwww.lexjansen.com/pharmasug/2011/cc/pharmasug-2011-cc26.pdf

Brucken, Nancy. 2008. One -Step Change from Baseline Calculations, and Other DOW-Loop Tricks.
Available at: www.lexjansen.com/mwsug/2008/pharma/MWSUG-2008-P01.pdf
Chakravarthy, Venky. 2003. The DOW (Not that DOW!!!) and the LOCF in Clinical Trials.
Available at www2.sas.com/proceedings/sugi28/099-28.pdf

Dorfman, Paul. 2009. The DOW-Loop Unrolled.
Available at www.nesug.org/Proceedings/nesug09/bb/bb06.pdf

Droogendyk, Harry, arrays – data step efficiency support.sas.com/resources/papers/proceedings13/519-2013.pdf

Johnson, Jim. 2005. The Use and Abuse of the Program Data Vector (The Procs).
Available at www.lexjansen.com/pharmasug/2005/technicaltechniques/tt03.pdf

Kahanew, Dalia; SAS data step – compile, execution, and the program data vector www.lexjansen.com/nesug/nesug11/ds/ds04.pdf

Lew, James & Horstman, Joshua: anatomy of a merge gone wrong www.scsug.org/wp-content/.../SCSUG_2016_Horstman_Merge_Gone_Wrong.pdf

Li, Arther, essentials of the program data vector PDV : directing the aim to understanding the data step! support.sas.com/resources/papers/proceedings13/125-2013.pdf

MATRO, John I Do, I Do, I DoW! A look at SAS DO and DoW loops Ppt siodes avalable on the web Available at:*https://vasug.files.wordpress.com/2012/04/i-do-i-dow.ppt*

Mehatab, Mohamed: understanding data step processing using PDv
https://www.sas.com/.../User%20Group%20Presentations/.../Mehatab-DataStepPDV.pdf.

Miron, Tom, the secret life of the SAS data step SAS User Group International 90 www.sascommunity.org/sugi/SUGI96/Sugi-96-27%20Miron.pdf

Whitlock, Marianne; the program data vector as an aid to data step reasoning analytics.ncsu.edu/sesug/2005/IN09_05.PDF

ACKNOWLEDGMENTS

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 registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

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

Author: Russell Lavery, Contractor Bryn Mawr, PA
Email: Russ.Lavery@verizon.net

**********Appendix A**********

```
/******Example 1***********/
ODS _all_ close;
ods listing;

options nocenter ls=200;
run;
Proc sort data=sashelp.class
```

```
          out=Sorted_Class;
 by sex age;
run;

proc print data=Sorted_Class;
run;

Proc sql;
select avg(height), avg(weight)
  into :Avg_ht, :avg_wt
  from Sorted_Class
quit;
%put Avg_ht=&Avg_ht avg_wt=&avg_wt;


Data Example1 (drop= AverageHt AverageWt ) /debug ;
   Retain Retained Obs No  Ret N Zro obs no;
   array Vital(2) AverageHt AverageWt (&Avg_ht, &avg_wt);
   AboveSet="Un"||"usual";
  set sorted_Class(drop=age rename=(Name=StName)) end=eof NOBS=OBS;
  by sex;
  If mod(_n_,2)=0 and sex="F" then LagName=lag1(StName);
    if _n_=1 then Obs_No=0; Else Obs_No=Obs_No+1;
    if _n_=1 then Ret_N_Zro_obs_no = 0;
       Else Ret_N_Zro_obs_no=Ret_N_Zro_obs_no+1;
    Retained_Obs_No=Retained_Obs_No+1;

    Simple_Obs_No+1;
  ArrayMin=LBound(Vital);
  ArrayMax=HBound(Vital);
  ArrayDim=Dim(Vital);
  Ht_Diff=height-vital(1);
  Wt_Diff=weight-AverageWt;
  IF _N_=1 THEN CALL SymPutX("ObsCnt",obs);
  run;

proc print data=Example1;
run;
  %put **&ObsCnt**;

Data _null_;
   if 0 /*this is always false & NEVER executes*/
       then set SasHelp.Air Nobs=ObsCount;
  call SymPut('OldObsIn_Air', left(Put(ObsCount,2.0)));
  call SymPutX('ObsIn_Air', ObsCount);
run;
%put OldObsIn_Air=**&OldObsIn_Air**;/*intentional error left(Put(ObsCount,3.0))*/
%put ObsIn_Air   =**&ObsIn_Air**  ;

/******* get # of obe into a macro ************/
Data _null_;
   if 0 /*this is always false & NEVER executes*/
      then set SASHelp.class NObs=ObsCount;
  call SymPutX('ObsInFile', ObsCount);
run;

%put 'ObsInFile' = **&ObsInFile**;


/******Example 2**********/
data Demog;
```

```
infile  datalines truncover firstobs=3;
input @1 name $char5. @7 ID 1. @10 age 2. @13 weight 3.;
datalines;
         1         2
12345678901234567890
Russ  1  23 170
Yatzi 2  28 101
Mike  3  29 240
;
run;
proc print data=Demog;
run;

Data Visits;
infile  datalines truncover firstobs=3;
input @1 ID 1. @5 Visit 2. @10 Systolic 3.;
datalines;
         1         2
12345678901234567890
1    1    120
1    2    110
1    3    105
2    1    100
2    2    100
3    1    140
;
run;
proc print data=Visits;
run;

data OOps/Debug;
      merge visits(SortedBy= ID) Demog (SortedBy= ID)
            ;
      by ID;
      label age = "Age in Months";
      Age = Age*12;
run;

proc print data=oops;
run;

data OOps2;
      merge Demog (SortedBy= ID)
            visits(SortedBy= ID);
      by ID;
      label age = "Age in Months";
      Age_mo = Age*12;
run;

proc print data=oops2;
run;


/******Example 3 ***************************************/
/*****subsetting where happens before the PDV***********/
/*****subsetting If happens in the PDV*****************/

Data Use_where /debug;
  set sorted_Class;
  where sex="M";
  run;
```

```
Data Use_if /debug;
  set sorted_Class;
  if sex="M";
  run;

 /**** example 4 double where from Jim Johnson*******/
data Oddwhere;
   set sorted_Class;
   where age GT 13;
   where sex="F";
run;

 proc print data=Oddwhere;
  run;


/***** Example 6 DOW Loop **/
 DATA Stu_Grades;
 infile datalines firstobs=3 truncover;
INPUT @1 name $char5. @7 sec $char1. @9 grade 3.;
DATALINES;
         1         2
12345678901234567890
Aruna f  90
Aruna f  93
Zihui f  97
Zihui f  95
Zihui f  95
;
PROC SORT DATA=Stu_Grades;
BY name;
run;
PROC PRINT DATA=Stu_Grades; RUN;

DATA Stu_Averages;
 Tests_taken = 0;
 Grade_total = 0;
DO UNTIL (LAST.name);
    SET Stu_Grades;
    BY name;
    Tests_taken + 1;
    Grade_total + grade;
END;
Avg_Grade = Grade_total / Tests_taken;
run;
 PROC PRINT DATA=Stu_Averages; RUN;
```