Paper 2120-2015

Inside the DATA Step: Pearls of Wisdom for the Novice SAS Programmer

Joshua M. Horstman, Nested Loop Consulting, Indianapolis, IN
Britney D. Gilbert, Juniper Tree Consulting, Porter, OK

ABSTRACT

Why did my merge fail? How did that variable get truncated? Why am I getting unexpected results? Understanding how the DATA step actually works is the key to answering these and many other questions. In this paper, two independent consultants with a combined three decades of SAS programming experience share a treasure trove of knowledge aimed at helping the novice SAS programmer take their game to the next level by peering behind the scenes of the DATA step. We'll touch on a variety of topics including compilation vs. execution, the program data vector, and proper merging techniques, with a focus on good programming practices that will help the programmer steer clear of common pitfalls.

INTRODUCTION

While SAS may at first seem to be just another programming language, it is actually quite different from most others. Both new programmers and those experienced in other programming languages can be tripped up by some of the subtler nuances of SAS programming. This paper aims to give those users a jump start by focusing on aspects of SAS software that are fundamental to its use but perhaps not terribly intuitive.

It is our hope that the reader can avoid some of the confusion and frustration that many SAS programmers face when they start moving beyond the most basic tasks in SAS programming. We present a series of “pearls of wisdom” obtained through experience. Each one can stand on its own. Taken together, they provide a solid grounding for the novice programmer looking to move to the next level. In each case, we provide a high-level overview and refer the reader to additional resources when more depth is required.

PEARL #1: THE BIG PICTURE

It will be helpful for the reader to first grasp the big picture of how a SAS program runs. This is a fairly complex subject, much of which is beyond the scope of this paper. We will limit our discussion to a broad framework which will provide the necessary context for proper understanding of some of the topics discussed later in this paper.

A SAS program is a sequence of one or more steps. Each step is either a DATA step or a PROC step. SAS runs them one at a time, completing one step before moving on to the next. The PROC steps are pre-built procedures which provide various ways to analyze and present data, while a DATA step is something crafted by the programmer to accomplish a particular task. Our focus here on what happens during the DATA step, which can be the source of much confusion.

When a DATA step is run, whether in batch or interactive mode, there are two distinct phases to the process: compilation and execution. Recognizing what occurs during each phase is key to explaining some of the less intuitive idiosyncrasies of SAS programming.

During the compilation phase, SAS takes its first pass through the DATA step code. The purpose of the compilation phase is to translate the SAS code into machine code which will subsequently be executed. During compilation, SAS parses the code to check for correct syntax and to setup the program data vector.

The program data vector, or PDV, is a temporary area in memory which SAS will use during the execution phase. Values which are read in from input data sets or are created during the execution of the DATA step will be stored in the PDV. The DATA step code will interact with the values stored in the PDV. At the end of DATA step execution, the values on the PDV will be written to the output data set.

More will be said about the construction of the PDV later in this paper. For now, it is enough to remember that the PDV is constructed during the compilation phase and then used during the execution phase. To dig deeper into this important topic, refer to Johnson (2012), Li (2013), and Whitlock (2006).

PEARL #2: VARIABLE LENGTH

With the above overview of DATA step processing in mind, we can now turn to a topic which has vexed many a new SAS programmer – variable lengths. Consider the two DATA steps shown below. Both make use of the SASHELP.CARS data set, which is a sample file included in most installations of SAS. The logic is identical and should produce the same results, but a subtle difference in the code produces an unintuitive result.

```
data cars1;
      set sashelp.cars;
      if msrp >= 20000 then price = 'EXPENSIVE';
      else price = 'CHEAP';
run;


data cars2;
      set sashelp.cars;
      if msrp < 20000 then price = 'CHEAP';
      else price = 'EXPENSIVE';
run;
```

CARS1

MAKE	MODEL	MSRP	PRICE
Acura	MDX	$36,945	EXPENSIVE
Ford	Taurus SE	$22,290	EXPENSIVE
Honda	Element LX	$18,690	CHEAP
Pontiac	Vibe	$17,045	CHEAP
Subaru	Outback	$23,895	EXPENSIVE

CARS2

MAKE	MODEL	MSRP	PRICE
Acura	MDX	$36,945	EXPEN
Ford	Taurus SE	$22,290	EXPEN
Honda	Element LX	$18,690	CHEAP
Pontiac	Vibe	$17,045	CHEAP
Subaru	Outback	$23,895	EXPEN

Figure 1. CARS Example

Notice that the values of PRICE appear as expected in the CARS1 data set but the CARS2 dataset contains "EXPEN" instead of "EXPENSIVE". An examination of the data set attributes will show that the PRICE column is a character variable of length 9 in the CARS1 data set but instead has a length of only 5 in CARS2. This can be seen by viewing the data set properties in interactive SAS, by viewing the output of PROC CONTENTS, or by using PROC SQL to inspect the contents of the DICTIONARY.COLUMNS table, as shown below.

```
proc sql;
      select libname, memname, name, type, length
            from dictionary.columns
            where upcase(libname)='WORK' and upcase(name)='PRICE';
      quit;
run;
```

```
Library
Name      Member Name                    Column Name                    Column    Column
                                                                        Type      Length
-------------------------------------------------------------------------------------------
WORK      CARS1                          price                          char           9
WORK      CARS2                          price                          char           5
```

Why would these two variables have different lengths when the code that created them is essentially the same? The key to answer this question is understanding how SAS builds the program data vector (PDV) during the compilation phase described earlier in this paper.

Recall that the PDV is a location in memory in which SAS will construct the output data set row by row. During the compilation phase, SAS builds the PDV by examining the SAS code which was submitted, not the data itself. When compiling the PDV for the CARS1 data set, the first statement processed is the SET statement which tells SAS that the SASHELP.CARS data set will be read in, so the PDV needs to contain a variable corresponding to each variable in the SASHELP.CARS data set and having the same attributes.

Next, SAS reads the IF statement. This statement references two variables: MSRP and PRICE. Since MSRP is on the input data set, SAS has already included MSRP in the PDV. However, this is the first time the compiler has encountered a variable called PRICE, so it adds a new location for it in the PDV. SAS sees that PRICE is assigned a value of 'EXPENSIVE', which is a string of 9 characters. Based on that, SAS designates PRICE as a character variable with a length of 9. When the compiler eventually reaches the ELSE statement, it sees that there is already a variable called PRICE on the PDV, so no further changes are made.

The important point here is that SAS sets the length of PRICE based on the value being assigned to it in its first occurrence in the DATA step. In the case of the CARS2 data set, a value of 'CHEAP' is assigned, so SAS gives PRICE a length of only 5. SAS does not look ahead to see that there are additional assignment statements later, nor does it matter that the assignment statement is conditional. SAS is not executing the code at this point in time. It is merely scanning the code to identify the elements of the PDV.

It's worth noting that at no point during this process was an ERROR or a WARNING written to the log. The log output for the DATA steps that created CARS1 and CARS2 are virtually identical. Thus, it is essential for the SAS programmer to understand the process described above in order to avoid frustrating errors and unexpected results that can be difficult to debug.

A simple solution to this problem is to include a LENGTH statement in the DATA step. The LENGTH statement instructs the compiler what length to give a particular variable in the PDV. It does nothing at execution time. Naturally, the LENGTH statement must occur prior to our first assignment statement so that the compiler will encounter it first.

```
data cars2;
      set sashelp.cars;
      length price $9;
      if msrp < 20000 then price = 'CHEAP';
      else price = 'EXPENSIVE';
run;
```

This a very simple example. The situation can be much more complex if there are multiple input data sets being merged or if the value being assigned to a new character variable is the result of a built-in SAS function. For additional information about the way in which SAS determines character variable lengths, refer to Whitlock (2006).

PEARL #3: VARIABLE ORDERING

Another topic that is often confusing to novice SAS programmers is how SAS determines the order of the variables that make up the dataset. Once again, understanding how the program data vector (PDV) is constructed during the compilation phase as discussed earlier is the key.

The order in which variables appear on an output data set will be the same as the order in which those variables appear on the PDV, although the PDV may also contain additional variables that are not written to the output data set (because of a DROP or KEEP statement, for example). As the PDV is constructed during compilation, variables are added to it in the order in which they are encountered by the compiler.

Consider the DATA step code above (the modified version near the end of Pearl #2) which constructs the CARS2 data set. As the compiler parses the syntax, the first thing encountered is a SET statement which reads in the data set SASHELP.CARS. At this point, all of the variables from SASHELP.CARS are placed on the PDV for CARS2 in the same order in which they appear in the input data set.

Next, the compiler comes to the LENGTH statement which includes a variable called PRICE. This is a new variable which does not yet exist on the PDV, so it is added. Since it is being added after the variables read in from SASHELP.CARS, it will appear after these variables in the PDV and hence in the output data set. In other words, when the CARS2 data set is viewed on screen using one of the many available tools (VIEWTABLE, SAS Universal Viewer, Enterprise Guide, etc.), PRICE will appear to the right of the other variables.

Armed with this knowledge, the savvy SAS programmer can now manipulate the program data vector to achieve a desired result. If we had wanted PRICE to instead appear as the first variable in the output data set, we could modify the program code so that the compiler will encounter it first. In this case, we can simply move the LENGTH statement above the SET statement. Since the LENGTH statement does nothing during the execution phase, moving it elsewhere in the DATA step will not change the values in the output data set at all. It will only affect the order in which the variables appear.

What if we would like to gain more control over the PDV and change the order of the variables which were read in from SASHELP.CARS? There are several ways to do this. One of the simplest and most common is through the use of a RETAIN statement placed at the top of the DATA step. If the variables are listed in the desired order on a RETAIN statement at the top of the DATA step, it will be the first thing encountered by the compiler as it builds the PDV. Thus, it will have the desired effect of making those variables listed the first variables on the output data set.

However, this method must be used with caution. The RETAIN statement can alter the output of a DATA step because it allows values from one record to be carried into the next record. Variables read in using a SET, MERGE, MODIFY, or UPDATE statement are automatically RETAINed anyway. Newly created variables and those read in using an INPUT statement are initialized to missing for each record. Including such a variable on the RETAIN statement for the purpose of reordering the PDV will also eliminate this initialization.

Another method for reordering the variables in the PDV involves the use of the ATTRIB statement. The ATTRIB statement is used to modify the labels, formats, informats, and/or lengths associated with one or more variables in a dataset. Since it is not an executable statement, it can be placed anywhere within the DATA step. By strategically placing the ATTRIB statement first (prior to statements like SET or MERGE that read the input dataset), we ensure that the compiler will encounter the variables on the ATTRIB statement first as it builds the PDV. While this process avoids some of the subtle complexities of the RETAIN statement, it is only useful when you actually want to modify one or more attributes of each variable you wish to reorder (or you simply don't care about the attributes). The ATTRIB statement will generate an error if it is not given a valid attribute list.

PEARL #4: MERGING WITH CAUTION

One task that causes problems for many inexperienced SAS programmers is merging data sets together. This is a complex topic upon which much has been written. We will point out a few common misconceptions and then point the reader to resources which can provide a more thorough treatment of the subject.

The first misconception has to do with what happens when a variable (other than the BY variable(s)) exists on more than one of the data sets being merged. Many programmers believe that values from a data set named later on the MERGE statement will overwrite values from identically-named variables found in data sets named earlier in the MERGE statement. This is true in many cases, but this belief can result in unexpected results in some situations.

To understand what is truly going on, we need to revisit the discussion above regarding the RETAIN statement. As we mentioned there, variables read in from a MERGE statement are automatically RETAINed. In other words, these variables are not initialized to missing each time a record is written out. Rather, the values read in will persist on the PDV across multiple records until they are overwritten by another value.

To see how this can produce unexpected results, consider the following example. We wish to merge together the two data sets shown below in Figure 2.

BASELINE

SUBJID	AGE	WEIGHT
1	35	190
2	48	175

VITALS

SUBJID	VISIT	WEIGHT	PULSE
1	1	188	60
1	2	191	57
1	3	193	58
2	1	177	72

Figure 2. VITALS Example

Our goal is to create a merged dataset which will include the subject ID number, age, visit number, the pulse at each visit, and the weight. However, we wish to carry the baseline weight forward and disregard the weights recorded at subsequent visits. Thus, we will list the BASELINE data set after the VITALS data set on the MERGE statement so that the values of WEIGHT from BASELINE will overwrite the values coming from VITALS. Here is the code:

```
data merge1;
      merge vitals baseline;
      by subjid;
run;
```

The resulting data set is shown below in Figure 3.

MERGE1

SUBJID	VISIT	WEIGHT	PULSE	AGE
1	1	190	60	35
1	2	191	57	35
1	3	193	58	35
2	1	175	72	48

Figure 3. VITALS Example – Incorrect Result

A careful examination of the values in the WEIGHT column shows that the first and fourth rows came from the BASELINE data set while the second and third rows came from the VITALS dataset. It is crucial for the professional SAS programmer to understand why this happened.

The explanation goes back to our previous discussion of the program data vector (PDV) and the implied loop of the DATA step. During the first iteration of the DATA step, the weight of 188 is read from the first record of VITALS into the PDV and is then overwritten by the 190 read in from BASELINE. At the end of the DATA step, the value is 190 is still in the PDV and is written to the first record of the output data set. The 190 is still retained in the PDV as well since all variables read in on a MERGE statement are automatically RETAINed.

During the second loop of the DATA step, the second record is read from VITALS and a value of 191 overwrites the 190 that is still in the PDV. However, there are no more records in the BASELINE data set for the current BY group (SUBJID = 1), so no value is read. At the end of the second iteration, the value in the PDV is 191 and that is what is written to the output data set.

Execution continues in this manner. No more records can be read from BASELINE until the next BY group begins, which occurs during the fourth iteration of the DATA step. At that point, the fourth record from VITALS is read with a value of 177, followed by the second record from BASELINE with a value of 175. At the end of the step, the PDV contains 175 and that is written to the final record.

This was not what we intended when we wrote the code. A simple way to solve this is to drop the weight from the VITALS data set by using the DROP= data set option on the MERGE statement as follows:

```
data merge2;
      merge vitals(drop=weight) baseline;
      by subjid;
run;
```

This produces the desired result shown in Figure 4.

MERGE2

SUBJID	VISIT	PULSE	AGE	WEIGHT
1	1	60	35	190
1	2	57	35	190
1	3	58	35	190
2	1	72	48	175

Figure 4. VITALS Example – Corrected Result

As you can see, you cannot always rely on variables being overwritten by variables having the same name on a data set listed later in the MERGE statement. To make your code more robust, eliminate or rename variables as needed to avoid duplicate same-named variables (other than those on the BY statement). In any case, having a thorough understanding of how the DATA step works will serve the programmer well.

This is just one of the ways in which a data set merge can produce unexpected results. To explore other merge traps waiting to ensnare the unsuspecting programmer, refer to Foley (1997) or Lew and Horstman (2013).

PEARL #5: THE OUTPUT STATEMENT

Every SAS programmer who has ever written a DATA step has used the OUTPUT statement, although they may not know it. The OUTPUT statement tells SAS to write the contents of the Program Data Vector (PDV) as a new record of the output data set. If a DATA step does not contain an explicit OUTPUT statement, SAS adds an implied one at the end of the DATA step. It is this implied OUTPUT statement that is responsible for the default behavior of writing out a record at the end of each loop through the DATA step.

However, there are many situations where it is useful to override this default behavior. You may wish to write the output record at some point before the end of the DATA step, or to write out multiple records for each iteration of the DATA step. In other situations, you might want to write an output record only when certain conditions are met. You might even need to write to multiple output data sets from the same DATA step. The savvy SAS programmer can accomplish these things and much more through the clever use of the OUTPUT statement.

Let's look at a simple example which highlights the strength and versatility of the OUTPUT statement. Consider the physical exam data set PHYSICAL shown below in Figure 5.

PHYSICAL

SUBJID	TEST	RESULT
1	WEIGHT	85
1	HEIGHT	185
2	WEIGHT	93
2	HEIGHT	172
3	WEIGHT	112

Figure 5. PHYSICAL Example: Input Data Set

Notice that this data is in a normalized ("tall and skinny") structure with one row for each subject for each test. Our goal is to add an additional row in which the subject's body mass index (BMI) is computed based on their weight and height. One way to do this would be to use PROC TRANSPOSE to get each subject's weight and height on a single record, compute the BMI, and then use PROC TRANSPOSE again to return the data to the original structure. Observe below how the use of the OUTPUT statement allows to perform this operation in a single DATA step.

```
data physical_plus_bmi;
      set physical;
      by subjid;
      retain weight height;
      if first.subjid then do;
            height=.;
            weight=.;
      end;
      if test='WEIGHT' then weight=result;
      if test='HEIGHT' then height=result;
      output;
      if last.subjid and not nmiss(weight, height) then do;
            test='BMI';
            result = round(weight / ((height/100)**2), 0.1);
            output;
      end;
      drop height weight;
run;
```

First, notice that this data step includes a BY statement even though it is not a merge. This has the effect of creating the FIRST.SUBJID and LAST.SUBJID variables which are used to indicate whether the current record is either the first or last record of the BY group corresponding to a particular value of SUBJID. Note also that this requires the input data set to be sorted by the BY variable. See Choate and Dunn (2007) for more on this functionality.

Secondly, notice the use of the RETAIN statement to allow values of WEIGHT and HEIGHT to persist across multiple iterations of the DATA step. Without the RETAIN statement, these variables would have been initialized to missing at the beginning of each iteration. However, we still want them to be initialized to missing when we encounter a new value of SUBJID (so that data from a previous subject isn't carried forward), so this is done manually.

Finally, observe that there are two occurrences of the OUTPUT statement in this DATA step. This first is unconditional and therefore executes with each iteration of the DATA step. This ensures that all of the original records are written to the output data set. The second one is conditional and occurs only when a BMI is calculated, which can be done only when a non-missing WEIGHT and HEIGHT are present for a given subject. If these conditions are met, the values of TEST and RESULT are updated and an additional record is written to the output data set. Figure 6 below shows the resulting output data set.

PHYSICAL_PLUS_BMI

SUBJID	TEST	RESULT
1	WEIGHT	85
1	HEIGHT	185
1	BMI	24.8
2	WEIGHT	93
2	HEIGHT	172
2	BMI	31.4
3	WEIGHT	112

Figure 6. PHYSICAL Example: Output Data Set

As mentioned earlier, the OUTPUT statement can also be used to write to multiple output data sets. Each output data set must also be listed on the DATA statement at the beginning of the DATA step. For example, recall the SASHELP.CARS data set from Pearl #2 above. Suppose rather than adding a PRICE variable to indicate whether the car is cheap or expensive, we wish to split the records into two data sets on the same basis. The following code could be used to accomplish this.

```
data cheap expensive;
      set sashelp.cars;
      if msrp < 20000 then output cheap;
      else output expensive;
run;
```

The result of this code will be two data sets: one called CHEAP containing all of the records from CARS with retail prices under $20,000 and another called EXPENSIVE containing those with prices of $20,000 or more. Each record is written to one or the other. Of course, it need not be the case that each record only goes to one of the output data sets. This is demonstrated in the following code.

```
data domestic foreign heavy light;
      set sashelp.cars;
      if origin = 'USA' then output domestic;
      else output foreign;
      if weight > 3500 then output heavy;
      else output light;
run;
```

The above code creates four output data sets. Each record from the input data set is either written to the DOMESTIC data set or the FOREIGN data set, and then is also written out to either the HEAVY data set or the LIGHT data set. Notice that even after a record has been written to the output data set, the contents remain on the Program Data Vector (PDV) and are available for further programming within the same iteration of the DATA step.

For additional information about the use of the OUTPUT statement, refer to pages 164-169 of the SAS Institute publication entitled Step-by-Step Programming with Base SAS® Software (2001).

PEARL #6: THE DANGERS OF AUTOMATIC TYPE CONVERSION

Variable types can be another source of confusion and problems for the novice SAS programmer. Every variable in a SAS dataset is either a numeric variable or a character variable. The variable type for each variable is determined during the compilation phase while the PDV is being constructed, and that type does not change during DATA step execution.

That seems fairly straightforward, but where programmers can easily run into trouble is when assigning numeric data to a character variable or character data to a numeric variable. While SAS gives the programmer several ways to convert data between numeric and character, SAS will automatically convert data as needed to attempt to execute the code as written. This is truly a two-edged sword. While it can provide a convenient shortcut for the programmer, sloppy and careless use of automatic type conversion can lead to unexpected or incorrect results.

To illustrate this point, let's look at an example. Consider the dataset shown below in Figure 7 containing several ten-digit U.S. telephone numbers stored in a numeric variable called PHONE.

DIRECTORY

NAME	PHONE
KIRK	8774775807
SPOCK	4034852994
MCCOY	4078555452

Figure 7. DIRECTORY Example: Input Data Set

Suppose we wish to create a new dataset containing an additional variable called AREACODE and containing the first three digits of each telephone number. An inexperienced SAS programmer might attempt to perform this task using the following code:

```
data directory2;
      set directory;
      areacode = substr(phone,1,3);
run;
```

Notice that the SUBSTR function requires a character variable as its first argument, but we have given it a numeric variable instead. Rather than generating an error, SAS happily converts the values of PHONE to character data so that the SUBSTR function can proceed to return the first three characters as requested. Observe the results:

DIRECTORY2

NAME	PHONE	AREACODE
KIRK	8774775807	8
SPOCK	4034852994	4
MCCOY	4078555452	4

Figure 8. DIRECTORY Example: Incorrect Result

The results are not what we were expecting. Instead of the new variable AREACODE containing the first three digits of PHONE, it contains only the first digit. In fact, although it is not apparent at a glance, it contains two leading spaces followed by the first digit.

In order to understand what has gone wrong here, one needs to know how SAS performs automatic type conversion. In the case of conversion from numeric to character data, there is an implied PUT statement writing the numeric data with the default format of BEST12. When a ten-digit integer is written with the BEST12 format, the result is a right-justified character string of length 12 with 2 leading spaces. That string serves as the input to the SUBSTR function, which returns the first three characters, exactly as specified.

The good news is that this problem is both easily detected and corrected. SAS automatically includes a note in the SAS log whenever an automatic type conversion is performed. In this case, that note reads as follows:

```
NOTE: Numeric values have been converted to character values at the places given by:
      (Line):(Column).
      3:23
```

The careful programmer will always scan the SAS log for notes such as these, investigating each one to ensure the code is producing the desired results. Your authors humbly suggest that a better programming practice is to avoid this situation entirely by explicitly handling all data type conversions using the appropriate SAS functions. Log notes such as the one above can almost always be prevented through careful programming. Below is one simple way it could have been handled in our example. There are, of course, many others.

```
data directory2;
      set directory;
      areacode = substr(put(phone,10.),1,3);
run;
```

This is just one of many situations in which automatic type conversion produces unexpected or undesirable results. As a matter of good programming practice, it is better to explicitly make all type conversions using the PUT and

INPUT functions with appropriate formats and informats. This leaves the programmer in complete control of the execution of their code. Slaughter and Delwiche (1997) sum it up nicely: "If a variable needs to be converted, you should do it yourself, explicitly, so there are no surprises."

Furthermore, as was noted above, automatic type conversion generates a note in the SAS log, but not a warning or an error. Relying on automatic type conversion requires the programmer to expect such notes and ignore them when reviewing the SAS log. This puts the programmer in a vulnerable position as each such note much be investigated to determine if it is expected or is the result of some other coding error or data issue that should be addressed. Under such circumstances, the programmer is at an increased risk of overlooking an actual problem. Instead, code should be designed to generate the cleanest possible log so that legitimate issues are more readily identified.

Section 3 of Gilsen (2007) is a good resource for further reading on this topic.

CONCLUSION

Understanding how SAS works "underneath the hood" is essential for the SAS programmer. Mastery of the concepts discussed in this paper will allow the SAS professional to avoid common errors and quickly diagnose and resolve problems that occur. We've only scratched the surface here. There is much more to be learned. The papers referenced within are a good starting point.

REFERENCES

Choate, Paul and Toby Dunn. "The Power of the BY Statement." Proceedings of the SAS® Global Forum 2007 Conference. Cary, NC: SAS Institute Inc., 2007. Paper 222-2007. http://www2.sas.com/proceedings/forum2007/222-2007.pdf

Foley, Malachy J. "Advanced Match-Merging: Techniques, Tricks, and Traps." Proceedings of the Twenty-Second Annual SAS® Users Group International Conference. Cary, NC: SAS Institute Inc., 1997. Paper 39. http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER39.PDF

Gilsen, Bruce. "More Tales from the Help Desk: Solutions for Common SAS® Mistakes", section 3. Proceedings of the SAS® Global Forum 2007 Conference. Cary, NC: SAS Institute Inc., 2007. Paper 211-2007. http://www2.sas.com/proceedings/forum2007/211-2007.pdf

Johnson, Jim. "The Use and Abuse of the Program Data Vector." Proceedings of the SAS® Global Forum 2012 Conference. Cary, NC: SAS Institute Inc., 2012. Paper 255-2012. http://support.sas.com/resources/papers/proceedings12/255-2012.pdf

Lew, James and Joshua Horstman. "Anatomy of a Merge Gone Wrong." Proceedings of the Pharmaceutical Industry SAS Users Group (PharmaSUG) 2013 Conference. Chicago, IL: PharmaSUG, 2013. Paper TF22. http://www.pharmasug.org/proceedings/2013/TF/PharmaSUG-2013-TF22.pdf

Li, Arthur. "Essentials of the Program Data Vector (PDV): Directing the Aim to Understanding the DATA Step!" Proceedings of the SAS® Global Forum 2013 Conference. Cary, NC: SAS Institute Inc., 2013. Paper 125-2013. http://support.sas.com/resources/papers/proceedings13/125-2013.pdf

SAS Institute Inc. Step-by-Step Programming with Base SAS® Software. Cary, NC: SAS Institute Inc., 2001. "Conditionally Writing Observations to One or More SAS Data Sets", pp. 164-169. http://support.sas.com/documentation/cdl/en/basess/58133/PDF/default/basess.pdf

Slaughter, Susan J. and Lora D. Delwiche. "Errors, Warnings, and Notes (Oh My): A Practical Guide to Debugging SAS® Programs", pp. 3-4. Proceedings of the Twenty-Second Annual SAS® Users Group International Conference. Cary, NC: SAS Institute Inc., 1997. Paper 68. http://www2.sas.com/proceedings/sugi22/BEGTUTOR/PAPER68.PDF

Whitlock, Ian. "How to Think Through the SAS® DATA Step." Proceedings of the Thirty-first Annual SAS® Users Group International Conference. Cary, NC: SAS Institute Inc., 2006. Paper 246-31. http://www2.sas.com/proceedings/sugi31/246-31.pdf

CONTACT INFORMATION

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

Joshua M. Horstman
Nested Loop Consulting
josh@nestedloopconsulting.com
317-815-5899

Britney Gilbert
Juniper Tree Consulting, LLC
Britney.Gilbert@JuniperTreeConsulting.com
www.JuniperTreeConsulting.com
@JuniperTree19