Paper BB-13-2011

Analyst Beware: Five Dangerous Data Step Coding Traps

David H. Abbott
Veterans Affairs Health Services Research
Durham, NC

ABSTRACT

The SAS® data step is a powerful data processing language with many features and abundant flexibility. However, it is complex and the results obtained from a data step can easily differ from what is intended and in some cases SAS provides no indication of anything amiss. This talk examines five dangerous traps in data step programming. These traps/pitfalls are dangerous because they: 1) produce no ERRORs or WARNINGs in the SAS log, 2) generate erroneous results, and 3) are likely to be encountered in common data step programming tasks.

The five traps occur in four different areas of data step programming:

- Missing value handling (1)
- FIRST/LAST variable related (2)
- Merging datasets (1)
- Character string handling (1)

These five traps do not comprise a comprehensive list of the dangerous traps in data step coding, but they are perhaps the most interesting and the most likely to lead to erroneous material in final reports. Let the analyst beware.

INTRODUCTION

SAS Base V9.2 provides a powerful language for creating and manipulating tables of data, both character and numeric. It is a rich language with many features. This richness and power in the SAS data step programming language has some unintended consequences. Probably most importantly, it means there are lots of ways for analyts to get behavior very different from what they expect. Often, SAS complains via either an ERROR or WARNING and the programmer is alerted to the problem and so no real harm is done. However, there are certain especially troublesome SAS coding pitfalls that:

- Produce wrong/unintended results
- Do not issue a warning or error
- Are likely to be encountered by programmers.

This manuscript examines five of these dangerous coding traps/pitfalls:

1. **Comparing to "."** – This practice leads to mishandling missing values.
2. **Failing to RETAIN** – Often summary variables are RETAINED automatically but sometimes they are not and failing to explicitly RETAIN them in this cases leads to unwittingly generating incorrect values
3. **Using subsetting IF with FIRST/LAST** – These two SAS features should rarely if ever be used together and when they are results go very wrong
4. **MERGEs with repeated variable names** – When the files being merged share some of the same variable names special care is required with merging
5. **Default string lengths** - SAS silently guesses about the length of strings unless explicitly instructed otherwise. Those guesses can be "almost" but not *adequately* correct.

The set of topics touched upon for each trap includes:

- The nature of the data step coding error, explicated via an example
- The usual and/or potential results and consequences
- Circumstances where the opportunity for the error arise

- Protective rules and practices for analysts

The examples provided here are made as simple as possible so that errors are easily seen and understood. As a consequence they are to some extent contrived and unusual. However, analysts should bear in mind that these same opportunities for error do come up and do so in much more complicated contexts where a myriad of other details compete for the analyst's attention.

COMPARING TO "."

SAS provides support for the notion of missing values, but sometimes the best way to use these facilities is not so obvious and it is easy to be tripped up, especially by some of the more arcane features like "special missing values". This feature allows users to categorize their missing values into any of 26 categories represented as .A to .Z. This feature can be useful, but it does complicate testing for missing values.

The occasion to test for missing values can come up, for example when trying to get rid of them, for example:

DATA STEP

```
DATA ShelfCounts; /* exclude missing values*/
  INPUT shelf bookCnt;
  IF bookCnt = . THEN DELETE;
DATALINES;
 1       46
 2       53
 3       .A
 4       .
RUN;
```

INTENDED RESULT

Obs	shelf	book Cnt
1	1	46
2	2	53

ACTUAL RESULT

Obs	shelf	book Cnt
1	1	46
2	2	53
3	3	A

For those not experienced with special missing values, the next-to-last line of the input dataset and the last line of the actual result may look rather odd. What is that "A" doing there in a numeric field? It is a special missing value of the "A" category. Why is it "A" instead of ".A" as it is on input? This is perhaps because it facilitates SAS implementation. Having the input rep and output rep differ for special missing values is a little unusual, but not really a problem. The important question is why is the line with "A" for book count there at all? The intent of the data step is to delete missing values! The regular missing value (".") was deleted but not the special missing value. In SAS, regular and special missing values do not compare equal. Hence, a missing value that should have been filtered out was not. The impact

of catching just some of the missing values and not all of them varies according to the specifics of the analyst's application, but in general, there is negative impact.

Luckily, this awkward situation is readily remedied by the **consistent** use of the MISSING function rather than simply comparing to ".". For example,

```
IF MISSING(bookCnt) then DELETE;
```

Other situations where comparisons to missing come up include: assigning imputed values to missing values and recoding the categories of a variable (very common operation).
Given the existence and behavior of special missing values, the author recommends **NEVER comparing to ".",** unless wanting specifically and only to exclude the regular missing value.

FAILING TO RETAIN

One use of the SAS data step is to extract useful information about groups of observations, e.g., the number of members in each of several user groups. The SAS data step facilitates such computations with the language features: BY, FIRST/LAST, and SUM statements. However, not all useful reductions are sums. For example, the analyst may want to extract a string that concatenates the initials of each member in the group:

DATA STEP

```
DATA theMembers;
  LENGTH listForGroup $80;
  SET InDs; BY group;
  IF FIRST.group THEN DO;
    listForGroup="";
    members=0;
  END;
  listForGroup= CATX(',', listForGroup, initials);
  members+1;
  IF LAST.group THEN OUTPUT;
RUN;
```

INPUT DATASET

Obs	group	initials
1	1	dha
2	1	frt
3	1	yz
4	2	zbt
5	2	bdp
6	2	da

INTENDED RESULT

group	members	list For Group
1	3	dha|frt|yz
2	3	zbt|bdp|da

ACTUAL RESULT

group	members	list For Group
1	3	yz
2	3	da

The output for **count** of members is as expected. However, the attempt to concatenate the initials of group members failed utterly. The list turned out to be just the last member of each group! What can be wrong?

By default, all variables in the data step (the Program Data Vector, or PDV) are set to missing at the start of the processing of each observation. So, listForGroup gets set back to missing every time an input observation is read – not what you want. To change this behavior the analyst can add:

```
RETAIN listForGroup;
```

as the second line in the step and all will be well.

So why did the count of members work correctly? Why don't we need:

```
RETAIN members listForGroup;
```

SAS automatically retains variables used in SUM statements (i.e. "members+1;"). And that is the tricky part. Since summing works fine without the RETAIN, it is easy to think that other reduction operations will as well, for example, operations like MIN(), MAX(), STD(), building up strings, or even just the simple operation of assignment when some particular condition occurs. In this last case, if the variable assigned to is not RETAINED, the value of interest will be lost as the next observation is processed.

The protective rule is: if the data step uses FIRST/LAST and a reduction operation isn't done with a SUM statement then a RETAIN statement will definitely be needed.

USING SUBSETTING IF WITH FIRST/LAST

Again using the example of the analyst needing to extract a string that concatenates the initials of each member in the group, let's complicate it a bit by specifying that initials beginning with "m" or greater be ignored. Using the subsetting IF that should be easy enough to do:

DATA STEP

```
DATA theMembers;
 LENGTH listForGroup $21;
 RETAIN listForGroup;
 SET InDs; BY group;
 IF initials < "m";
 IF FIRST.group THEN DO;
   listForGroup="";
   members=0;
 END;
 listForGroup= CATX("|", listForGroup, initials );
```

```
  members+1;
  IF LAST.group THEN OUTPUT;
RUN;
```

INPUT DATASET

Obs	group	initials
1	1	dha
2	1	frt
3	1	yz
4	2	zbt
5	2	bdp
6	2	da

INTENDED RESULT

group	members	listForGroup
1	3	dha|frt
2	3	bdp|da

ACTUAL RESULT

group	members	listForGroup
2	4	dha|frt|bdp|da

So, what seemed so straight-forward a change to make has turned out terribly wrong with no hint of trouble in the SAS log. The fix is a ***one-word fix*** – just change the "IF initials" to "WHERE initials" and the intended result is achieved. How can this be? SAS marks observations as FIRST/LAST and the SUBSETTING IF discards some of these marked observations rendering the initial marking invalid and producing bizarre results. The rule to follow for protection is: ***do not combine the use of SUBSETTING IF and FIRST/LAST in the same data step.*** Usually the analyst can achieve the same effect with the WHERE statement and achieve better efficiency in the bargain.

MERGES WITH REPEATED VARIABLE NAMES

Many analysts use to use the MERGE statement of the data step frequently. It seems to be often the case that one dataset has some of the variables needed for a computation and another dataset has the rest. The MERGE statement combines the variables from multiple files. However, with the MERGE statement, it is all too easy for the unwary analyst to achieve unintended/incorrect results. Consider this example that merges information regarding certain medical procedures to pull together a table of the tests, the days required to get results, and the cost and priority.

DATA STEP

```
DATA MedProcDs;
  MERGE
    InDs1   /* vars are: test days cost */
    InDs2;  /* vars are: test days priority */
  BY test;
RUN;
```

INPUT DATASET ONE

test	Days to Results	cost
FOBT	2	40
colonoscopy	3	2700

INPUT DATASET TWO

test	Days till Repeat	priority
FOBT	365	2
colonoscopy	1825	1

INTENDED RESULT

test	Days to Results	cost	priority
FOBT	2	40	2
colonoscopy	3	2700	1

ACTUAL RESULT

test	Days to Results	cost	priority
FOBT	365	40	2
colonoscopy	1825	2700	1

In the **actual** result it looks like “Days till Repeat” values have transmuted into “Days to Results” values! Reporting to a superior that patients are waiting 5 years to get the results from their colonoscopies could get an analyst in trouble. So, it is important to understand the root cause and nature of this MERGE statement behavior.

As the reader may suspect, the root of the problem is that both InDs1 and InDS2 contain a variable named “days”, and the values in the “days” variable in the two files differ (in this case because their meanings differ). So, when the datasets are merged, how does SAS deal with a multiplicity of “days” variables? The two rules that apply are:

1. The **label** of the DAYS variable in the first merged dataset wins out; it is retained in favor of the labels from subsequent datasets.
2. The **value** of the DAYS variable in the last merged dataset wins out; it overwrites values from preceding datasets.

This behavior is confusing and sets up analysts to get unexpected results. It is best to avoid it by the prudent use of KEEP=, for example

```
DATA MedProcDs;
  MERGE
    InDs1
    InDs2( KEEP= test priority);  /* vars are: test days priority */
  BY test;
RUN;
```

Usually only one of the datasets in the MERGE statement should occur without a KEEP= option. However, sometimes both datasets have hundreds of variables and the count of variables needed from both datasets is large. In this case, using KEEP= is not practical. An alternative is to use DROP= to eliminate the undesired variables whose names conflict with desired variables. However, with hundreds of variables in each dataset, determining exactly which variables to put into the one or more DROP= lists is no simple task. The author has created a macro %OverlappedVars that greatly facilities the task by detailing the contending variables:

```
%macro OverlappedVars(
  listOfDs=, /* the datasets  vars scanned/compared */
  outDs=, /* the DS detailing the overlaps found */
  sp=olv_tmp, /* i.e. scratch prefix */
  cleanUp=1 /*if true deletes scratch datasets  */
);
/*
Invoke with:
   %OverlappedVars(listOfDs=, outDs= );

Determine the variables that occur in two or more of a
user supplied list ofdatasets. These variables must be
dealt with before a merge of these datasets can
be safely done.
*/
```

	Doubly Written Field	First Writer	Second Writer
1	b	tmp1	tmp2
2	b	tmp1	tmp3
3	c	tmp2	tmp3

DEFAULT STRING LENGTHS

Base SAS has a mixed approach to dealing with the length of character strings. On the one hand, character variables in Base SAS are required to have a specified maximum length and it is consistent policy to discard any extra characters when a longer string is assigned to a shorter one. On the other hand , Base SAS character string functions like LENGTH(), CATS(), CATX(), treat character strings more like variable length entities and SAS often deduces/guesses at an appropriate length for character string based on their implied length at first use, thereby relieving the analyst from explicitly specifying lengths. Hence analysts often get intended behavior without giving the lengths of character strings much thought. However, in some cases a “bad guess” about length by SAS and the SAS policy of silently discarding excess characters combine to produce entirely unacceptable results – consider this example involving the input of a list of state names, with the added wrinkle that the names to be uppercased and blank input lines to show up as “none”:

DATA STEP

```
DATA theStatesDs( DROP= theName);
  INPUT theName $ 1-16;
  IF missing(theName) then StateName="none";
  ELSE StateName = upcase(theName);
  DATALINES;
<<blank line>>
Texas
```

```
North Carolina
RUN;
```

INTENDED RESULT

```
 Obs    StateName

  1     none
  2     TEXAS
  3     NORTH CAROLINA
```

ACTUAL RESULT

```
 Obs    StateName

  1     none
  2     TEXA
  3     NORT
```

SAS very reasonably sets the length of theName to 16 and since StateName is based on theName one might expect that its length will be 16 as well. However, as the actual output demonstrates, the length of StateName variable is 4! Why? The "first use" rule strongly suggests that having the first line of data a blank line is the culprit. In this case, the literal "none" is length 4 and that is the first value assigned to StateName. Case closed?

No, it turns out that even if you swap the order of Texas and the blank line in the data you get the same result – the length of StateName is still 4! Perhaps when the data step is compiled the length 4 literal is noted and the length of the recipient of the literal (StateName in this case) is assigned length 4 even before any lines of data are processed. In any event, it is clear that it is difficult for the analyst to guess the length that Base SAS will assign by default to a character variable in all cases. The protective rules for the analyst are: 1) when at all in doubt explicitly set the length, and 2) always look at a sample of the data in the output datasets that are created to confirm that character strings are not being truncated.

Some other examples of other situations where an analyst might run into this issue include:

- Whenever character literals are present in a data step and assigned to a character variable
- When the recipient variable is assigned values from more than one other variable which may have unequal lengths
- When building up a recipient string by repeated concatenation.

CONCLUSION

Some closing remarks regarding this topic are:

- SAS data step programming provides powerful data manipulation capabilities and usually ERROR and WARNING messages alert analysts to coding errors. However, the five coding traps presented are important exceptions to that rule.
- Good coding practices and awareness enable analysts to avoid these traps.
- Nevertheless, code inspection, output inspection, cross-checking, and thorough testing are not optional in data step programming; they are a must.

REFERENCES

Foley, Malachy J. 1998. “MATCH-MERGING: 20 Some Traps and How to Avoid Them.” Proceedings of the 23rd Annual SAS Users Group International Conference, http://www2.sas.com/proceedings/sugi23/Advtutor/P47.pdf

Tian, Yunchao 2007. “The Power and Danger of Automatic Retain.” Proceedings of Northeast SAS Users Group Conference, http://www.nesug.org/proceedings/nesug07/cc/cc42.pdf.

Galligan, Olena 2010. “Creating Variables: Traps and Pitfalls.” Proceedings of Western Users of SAS Conference, http://www.wuss.org/proceedings10/coders/2917_4_COD-Galligan.pdf.

ACKNOWLEDGMENTS

The views expressed in this paper are those of the author and do not necessarily reflect the position or policy of the Department of Veterans Affairs or the United States government.

Without the leadership and encouragement of Dr. Dawn Provenzale, director of the Durham Epidemiologic Research and Information Center at the Durham VA Medical Center, this work could not have occurred. She takes a strong interest in fostering many dimensions of excellence in her employees.

CONTACT INFORMATION

Name	David H. Abbott
Enterprise	Center for Health Services Research in Primary Care
Address	Durham Veterans Affairs Medical Center
	HSR&D Service (152)
	508 Fulton St.
City, State ZIP	Durham, NC 27705
Work Phone:	919-286-0411
E-mail:	david.abbott@va.gov