

10 Things I Hate in SAS Coding

SAS® Software software was designed from its inception to be easy to use and forgiving of poor or uninformed choices. Unlike other software of its era (like PL/I, upon which SAS software was based and was mostly written in) SAS was never a strongly typed language. One does not have to declare a variable in great detail before using it; in fact, the SAS Data step goes to great lengths to guess what data structure will be appropriate to the coder's needs.

In actual practice, the Data Step compiler tends to guess rather accurately. The talented developers at SAS Institute have had many years to hone and refine their craft, and the foundation upon which it is based is a sound one.

However, this does not mean that those who consider themselves (and whose managers consider) as professional programmers should have a indifferent attitude about how they write and format SAS code. In very many cases, code tends to outlive it's intended duration; how many people coding in the 1960's and 1970's could ever imagine that their programs would survive long enough to cause the Y2K uproar?

The purpose of this paper is to share a series of coding techniques which cause the author to wince, cringe, or curse when encountered. Art Carpenter has written a famous series of papers: "Programming for Job Security Revisited: Even More Tips and Techniques to Maximize Your Indispensability" (SUGI 23); "A Bit More on Job Security: Long Names and Other V8 Tips"(SUGI 26 paper 235-26). "Job Security: Using the Macro Language to Full Advantage" (PharmaSUG 2005). Unlike these papers, which clearly show dubious coding techniques while provoking a good belly laugh, the current paper attempts to provide justification for each preference; all based on personal experience, and some purely on personal preference. The purpose in all cases however is to utilize each code

segment as a teaching opportunity. Each case will be evaluated and rated on two independent (and arbitrary) scales; how likely the technique is to cause actual harm, and how personally offensive the author finds it. Note that while the later is completely subjective and open to great debate, the former is based on actual experience crafting software solutions using the SAS® System. Note that the author's history with SAS encompasses nearly three decades, and almost all platforms ever supported. Some of the cautions mentioned in this article may not cause any problem in modern environments (e.g. SAS Version 9.1.3 SP4); however if the problem has surfaced before it might well do so again in the future.

Case # 1: Clearing title statements. This is often coded as the keyword title followed by two single (or two double) quotes.

```
title1 '';
```

On the personal dislike scale, this barely registers as annoying. However, in some versions/platforms, this leaves a single quote mark as the text of the title, not the blank probably intended by the coder. The SAS documentation is quite clear in this respect; the proper definition of a null title is the keyword *title* and title number, followed by only a semicolon.

Case # 2: Multiple Titles. If having one title statement with poor syntax is unfortunate, how about 10? Consider:

```
title1;  
title2;  
...  
title10;
```

Note that the 10 possible title (as well as the footnotes) are hierarchical in nature. As soon as title *n* is defined, all titles with larger *n* are erased. Coding `title1;` erases all 10 title slots. Coding multiple statements is unnecessary.

10 Things I Hate in SAS Coding

Case # 3: Programatically clearing the Log. Many coders like to clear the SAS Log in the Display Management System between interactive runs. This can be a handy practice so as to ensure that the SAS notes being examined do represent the actual last code submitted. Something like:

```
dm 'clear out';  
dm 'clear log';
```

This does accomplish the desired end; however, if this same SAS code is run in batch mode instead of interactively, then an error is generated (the `clear` command clears the Log *window*, not the log *file*). Again, this does not impair the subsequent execution of any SAS code. But it leaves an extraneous error message; in a regulated business (such as the Pharmaceutical industry) having an `ERROR` message in the SASLOG is strictly forbidden by many companies. Consider that clearing the Log window is only an attribute of running interactive SAS, consider instead defining a function key:

```
clear log; clear out; submit
```

This will have the desired effect of clearing the contents of the two windows before submitting the current code. When the code is run in batch mode, no DM command will be present to generate an `ERROR`.

Case # 4: Confusing Length and Formats. Consider the following Data step statement:

```
length x $3.;
```

Very simple code like this occurs all the time. (Un)fortunately it actually does what the coder intended, but for the wrong reason. The `length` statement has only one purpose: directing the SAS compiler to create a field with a specified amount of storage. The absence of a dollar sign (\$) results in a default of a numeric field. The statement above seems to be requesting a character field named *x* be created, with length of 3 bytes. What's wrong?

The formal syntax of a `length` statement involves only 4 elements:

1. The keyword `length`, followed by
2. `variable(s)`
3. possible (\$)
4. semicolon (;) as an end of statement.

Nowhere in this syntax does the period (.) appear. The presence of a period (.) should be used when indicating a format or informat specification:

```
format x $3.;
```

As mentioned in the introduction, the SAS compiler is by design and by dint of much hard work very forgiving. SAS “knows” that you meant to ask for 3 character bytes, and graciously provides the same. It is not correct syntax, however.

Case # 5: Extra step boundaries. It is considered by many accomplished SAS coders that providing an explicit step boundary is a preferred coding practice. Usually this is accomplished by adding a `<run;>` statement after a Data step, and after a procedure step. However, certain SAS procedures support something called *run group processing*. This is where a grouped series of statements within the procedure run as a unit, with other subsequent grouped statements running later, but within the control of the same procedure execution. These procedures are then terminated explicitly by the use of the `This` can be tremendously useful tool. Consider the following code:

```
proc datasets library=work  
              nolist;  
  modify test;  
    rename x1 = x;  
run;  
  modify test;  
    label x = 'New Label';  
run;  
quit;
```

Note that the `run;` statement breaks the

10 Things I Hate in SAS Coding

procedure into two distinct parts: first, a particular data set [`work.test`] is selected for processing; then an existing field is renamed. As soon as the `run;` statement is encountered, the rename occurs. This allows the subsequent new label to be associated with the newly renamed field, without any confusion about whether to use the new name or the old name. The final `quit;` statement formally concludes the execution of the procedure.

Another variation on this theme is excluding a `run;` statement when one would be logically required. Consider the code where:

```
proc datasets data=work;
  delete dsetA;
quit;
```

Again, the code above might work just fine in the current version and platform of SAS that you test it on. However, on previous occasions this code has failed to provide the expected functionality (deleting a SAS data set from the work library), with no note, warning, or error message; simply no effect. Subsequently the author always codes at least one `run;` statement before the closing `quit;` statement.

The inverse is also true; some coders will place a `quit;` statement after procedure call. This wastes five keystrokes and is very mildly annoying, but does no harm.

Lastly on this point: there are only three types of statements that occur in a SAS program:

1. Data step statements
2. Procedure statements
3. Miscellaneous statements

There are very few of the latter, and they can occur between steps. Examples include:

- filename
- libname
- title
- footnote
- options
- goptions

These statements are all executed as soon as they are encountered by the SAS compiler; since they are not part of a step, they do not need any combination of `run;` or `quit;`.

```
title1 'This is title1';
run;
libname x 'location';
run;
```

Case # 6: Rectification of Names. It is very helpful to ensure that variable names in SAS data sets accurately reflect their meaning. In addition to avoiding nonsense names such as {vara something z}, also consider the following code:

```
proc means data=hr.salary;
  var annual;
  output out = work.salary
         mean = annual;
run;
```

Assume here that the existing permanent SAS data set [`hr.salary`] contains information about the individuals working for your organization. One would feel comfortable assuming that the field [`annual`] contains each worker's annual salary level. The `means` procedure will summarize the requested data set, and create a new data set in the work library, [`salary`]. This newly created data set will contain a new field also called [`salary`]. The problem is that the same name refers to two completely different types of information; one a collection of information about a group of people; and another one that represents the average of all those same people. Yes, they are both numeric fields; yes they represent information about annual income. However, their scope of meaning is dramatically different. When changing levels of organization like this, consider at least changing the output field name to be something distinct. Ideally, the new name (and label!) should indicate clearly what is being represented.

10 Things I Hate in SAS Coding

Case # 7: Reusing work data set names. This topic is somewhat tricky, as the technique is widely used and not intrinsically problematic. However, consider the following case:

```
data huge;
  set huge;
  length newvar $ 12;
  <define newvar...>
  drop critvar;
run;
```

As an incentive, the selected table name [huge] indicates a very large table, such that re-running the code steps which lead up to its creation would be time consuming and tedious. Now assume that after executing this long Data step, you discover that the allocated length of 12 bytes was insufficient to hold all the new values of [newvar], and truncation of data values has occurred. Now the code will be forced to re run all the procedure and Data steps which lead up to this final error.

Depending on your data characteristics and platform, this could entail hours of wasted time. Also, come critical variable [critvar] has been dropped out of the data set; rerunning the code now will fail because this field is no longer available for calculations.

Of course it might be considered a burden to think of many distinct names if a series of steps are required to perform slight changes in a data set. A common solution is to append a numeric suffix to a series of work data sets, producing for instance the series {huge1 huge2 huge3 huge4...}. This solves the problem being unable to re-run a specific step without re-running many previous steps. However, it might introduce a new problem: over-proliferation of temporary data sets. This is not a mechanical problem unless the data are so massive as to overwhelm the physical resources of your computing platform (more about this soon). One possible solution is to add data set labels to the work data sets, which help the coder keep track of how the data flow is managed. For instance,

```
data huge1(label='Initial
data');
data huge2(label='Calc
Critvar');
data huge3(label='Filter Out
Bad Values');
data huge4(label='Filter out
duplicate data');
```

It is perhaps an unusual technique to provide labels to data sets that, by definition, are ephemeral and will soon be deleted. It certainly is not necessary in programs with simple logic trains. However, using this technique when constructing very long, convoluted programs with many intricate steps helps a great deal in keeping track of when actions occur where in the data flow and the program logic.

Another trick to help capitalize on this technique is the use of a fairly obscure option:

```
options details;
```

This will then show more information than the default about SAS tables in the SAS Explorer window, including the actual number of rows and fields in each data set, as well as the data set labels. When the work library gets crowded and complex, this can go a long way towards clarifying what data is contained in which tables. One caveat is to be careful when using this technique and examining libraries which are on a remote host or server, or where network latency issues may slow down the retrieval of this additional metadata. In these cases, the time to refresh the Explorer window might become unacceptable, overwhelming the utility provided when examining the work library.

Case # 8: Choosing the wrong tool for the job. It is an old saying that “to a worker with only a hammer, everything looks like a nail.” One benefit (and problem!) with learning and using the SAS system is that there are numerous tools available. The answer to the question “Can this be done in SAS” is usually not answered simply “Yes”, but a more

10 Things I Hate in SAS Coding

complex “Well, there’s 6 different ways to do that; which is optimal?” Consider the small code fragment below:

```
data huge;
  set huge;
  rename
    x = y;
run;
```

This is perfectly legal syntax, and will indeed accomplish exactly what the code intends; in the data set [huge], the existing field [x] will instead be called [y]. Also, if the data set being manipulated was small in comparison to the abilities of your computing platform (say 6 fields and 50 observations) then using this suboptimal technique will hardly matter or even be noticeable as a problem. But if the data set (as implied by the selected name) is huge by the standards of your hardware (say measured in multiple gigabytes of storage), then this technique is truly unfortunate.

Remember that when running the Data step, in nearly all cases one is creating (or in this case re-creating) a SAS data table. Note in this code fragment that no other manipulations are occurring; no new fields being defined, no data filters being applied; no `keep` or `drop` statements which change the physical contents of the data. So in this case, we are employing the Data step to read and write potentially millions (or b- or t-) of records just to rename one field.

Remember also that SAS data tables consist of two tightly coupled parts; a data set header which contains all the metadata about the table, and the actual data values themselves.

In the SAS system these two parts are inseparable from one another (note we are not referring to data Views, which follow different rules). All we really need to do in this particular example is change a field name, or perhaps a label, or format specification. All of these characteristics are part of the metadata, and not the actual data values. While the Data step is perfectly capable of making these

desired changes, it does so at the expense of reading and writing every single data record in the table.

A much more efficient tool to solve these problems is `Proc Datasets`. The purpose of this “Swiss Army Knife” in the SAS system is precisely to perform the before-mentioned (and indeed many other) functions. The step above might instead be coded as:

```
proc datasets library=work;
  modify huge;
  rename x = y;
run;
quit;
```

So the exact same endpoint is achieved, and the syntax required is not particularly difficult. The benefit derived from using this technique is that SAS is only performing the work needed, which is to manipulate the metadata of the table, while avoiding the entire process of re-creating the table and all the attendant I/O. The same technique may be used to change data set labels, variable labels, variable formats and informats, and so on.

This is not to imply that `Proc Datasets` will always run instantly. It can also be used to create a data set index, which for a huge data table (hopefully not in the work library!) could take a considerable amount of time.

Case # 9: The Compiler is always right. The SAS system has been growing and evolving for decades now, and the amount of functionality it contains is astonishing (and sometime overwhelming!) One measure of this growth is the physical size of the official documentation as provided by SAS Institute. Starting with something not much larger than a graphic novel, the entire documentation set would now fill several bookcases. During this evolution, the SAS developers have worked hard to preserve older functionality while adding new abilities. In fact, this has been one of the major advantages of using SAS software.

10 Things I Hate in SAS Coding

However, strange things do occasionally creep in and out of the system. Consider the following comment block:

```
/* Don't forget to check */
/* the following data set */
/* for errors.          */
```

According to the official published rules of comments in the Data step, the code above is perfectly legal and appropriate. However, in a previous version of SAS running under Windows, suddenly the tokenizer started “seeing” the embedded singular quote mark, and production macro which had been running correctly for years suddenly crashed and burned. This was not even during an upgrade to a new version, but after applying a bundle of hot fixes. After this experience, the author now prefers to be just a little less lazy:

```
/* Do not forget to check */
/* the following data set */
/* for errors.          */
```

Case # 10: Debugging options. An old axiom is that any program complicated enough to be of any value is bound to have errors. Many applications of the SAS System are also bound up with data, which may induce errors when interacting with otherwise perfectly fine code. So it is entirely reasonable to find the following kinds of code inside a program which is under initial development or substantial revision:

```
options
  symbolgen
  mprint;

data ...;
  set ...;
  put x= y= z=;
run;

proc print data=temp01;run;

...

proc print data=temp99;run;
```

However, once a program has been thoroughly vetted (or validated, if you're in that business),

then these types of code segments should be removed, commented out, or placed into “dummy macros”:

```
%macro dl;

  proc print data=temp01;

  run;

%mend dl;

/* no call to this macro;
/* exists anywhere in the;
/* program;
```

Trying to look through hundreds of pages of SASLOG filled with expanded macro calls is an unwelcome exercise in frustration and patience.

Hopefully the tales of frustration and woe have matched some of the experiences of the reader, and the explanation shed some light on how to approach the art of SAS coding. SAS is an amazingly complex language with a long history; learning where some of the previous potholes were might help steer a course through smooth driving in the future.

Author contact:

Paul Hamilton
Amgen, Inc.
hamiltonp@amgen.com
(206) 265-7689