Paper 230-2012

Macro Coding Tips and Tricks to Avoid "PEBCAK" Errors

Matthew T. Karafa, PhD, Cleveland Clinic Foundation, Cleveland, Ohio, USA

ABSTRACT

No matter how well you document what a macro's parameters are supposed to accept, at least one user will ignore your hard work and try something unanticipated. This presentation describes some general tips and tricks I use wne writing macros from my parameter checking scheme, which provides such users direct feedback via the SAS® log to how I use comments to document code. Thus, it should prevent the macro author from having a long debugging session, only to conclude that the user passed the macro inappropriate information. This presentation should be applicable to both beginning and advanced macro coders alike.

INTRODUCTION

If you have been writing SAS macros for any length of time you most certainly have a library of personal usage macros. These are macros we never intended for anyone to want but ourselves, and make our day-to-day lives simpler. As many authors at past SAS® Users Groups meetings and SAS® Global Forums have said – it is just coding smarter rather than harder. These macros are often little documented beyond our own shorthand, and aren't going to convey exactly what is needed for the program to run correctly.

Then it happens – someone admires a little snippet you've written and asks you share it with them or to put it up in the company library. Now your simple little workhorse macro is venturing out into the wild and needs to work with other people's minds. No problem – thinks the coder – I'll just write out what each of the parameters expect and put it up for all to use. If you're like me – you send this on to the "powers that be" and think that's the end of it and go on patting yourself on the back.

All goes well, until an inexperienced user tries to pass the macro an incorrect parameter. The results that were supposed to be black are now white. "There is no way this code is correct" – thinks the inexperienced user – "that Karafa can't code his way out of a paper bag." Now the user of your workhorse is on the phone with you (or worse in your boss's office) and demanding to know why you have written a macro that generates such garbage. Suddenly you find yourself needing to check his work and if you're particularly unlucky all the other programs that this little workhorse! Yuck. All because the program never told this user that it wasn't designed to do what it just asked to do. The best way to prevent this situation from happening is smart coding. Wouldn't it be great if the program could auto-magically figure out if someone just asked for something outside the scope of what was programmed into the code? It would be nice if the program could be smarter than the inexperienced user and let them know that – indeed the macro is not wrong, but the user made a mistake. Parameter checking is the first step toward preventing these kinds of problems from happening. Or wouldn't it have been wonderful if there was only one place to look to change the value of some crucial constant that is used through out the macro. These are some of my bag of tricks I use when programs migrate to the wild to make sure this kind of scenario doesn't happen.

TRICK #0: DOCUMENT! DOCUMENT! DOCUMENT!

If you can't figure out by now it will soon become apparent that I write my code with the expectation that there will be periods where it works and I will not even consider what's happening to it – I'll just use it. In between these wonderfully quite periods of life, there will be times when you're screaming at the machine beating your head against the wall trying to quash niggling bugs that evade your ever watchful gaze. When there is such a lag between coding I can't stress enough the need to document, comment, remind and just generally slap yourself in the face with what you were doing the last time you delved into a given program. Finding an obscure piece of SAS code to produce the results you are looking for is typically a Google search away. Remembering ***WHY*** that obscure trick of SAS code is in your program will really give you the leg up to fixing it when it breaks!

My use of commenting has evolved over my career. I now typically use long-ish variable names in "CamelCase" to make sure I know what's in a given variable. I expressly use the mnemonic operators (eq, gt, ge, le, etc…) for comparison and only use "=" for assignment. There are more section/comment breaks in my code with outlines in notes as to where my head was when I was writing things. In collaborative programs I encourage "initials Date: " as the first line of each one. These things work for me, my coding style and my increasingly addled and disorganized brain – your mileage might vary. The point is find what works for you and the effort that puts it in balance and STICK with it. Coding style things – like the indenting case conventions and other non-essential tricks you will observe in this paper are just as important for your code to be understood as knowing where to put the semicolon. They are to your code like white space, paragraph headings and bold faced type are to a good text book.

TRICK #1: PARAMETER CHECKING

PROGRAMMING STRATEGY

The biggest problem with inexperienced users is that they don't know what not to use. If we as the programmer don't tell them something is invalid they never learn and are constantly coming to us to correct a non-existent problem. Thus the bulk of this paper will focus on my parameter checking strategy. If we use something akin to this basic parameter checking strategy our code looks cleaner and more professional, not to mention provides a better user interface. The strategy starts with the assumption that the user has entered all the parameters correctly – as un likely as that might be! This is done by setting a Boolean error flag to FALSE. Then we craft code to check each parameter in turn, and if it is in error we do 2 things: 1) change the error flag to TRUE; 2) provide an error message in the log using `%PUT`. Such error messages should contain a mention of the macro parameter that is in error as well as the condition that caused the error. If there are expected values, these should also be provided. If you're trying to understand how to write such an error message, look to the ones SAS® provides with the procedures, it's the same idea. Once all the parameters have been checked, if there are no errors we continue to the macro body. If there is a problem, the macro exits using the abort command.

GETTING STARTED

Before we can start checking we first create have to create a Boolean macro variable which is FALSE if there are no errors and TRUE if we have found any:

```
%local __Macro_Err;
%let __Macro_Err=0;
```

Note that I tend to doubly protect myself against macro name "stepping". First I use the `%LOCAL` command to define the scope of `&__Macro_Err.` as local to my macro. The scope defines for SAS® where the macro variable is stored as well as where it is available. Since it's local to the macro only, outside whatever macro created it `&__Macro_Err.` will not exist. This prevents it from being assigned the wrong value by another macro or being used incorrectly in another place. Second I tend to precede internal macro variables with underscores. This little trick makes it doubly unlikely that some unsuspecting user will step on the names in my macro. It also has the side effect of making it easier to identify them as I read through the code later. It is a little compulsive, but this little bit of compulsivity has saved me work in the long run, so I encourage the behavior. Now that we have our flag set, the simplest check is to make sure our required parameters have any value at all. Using the `%LENGTH()` macro function is the best way to check for this:

```
%if %length(&RequiredParm1.) = 0 %then %do;
    %put ERROR: NO VALUE SUPPLIED FOR RequiredParm1;
    %let __Macro_Err=1;
%end;
```

Certainly I could have used "`&RequiredParm1.=`" rather than the `%LENGTH()` macro function with the same results. My opinion is that the macro function method is easier to understand when returning to the program at a later date. Notice that the `%PUT` statement that provides the error messages starts with "ERROR:" just like a stock SAS® error message. This enables the Log system to highlight the error messages properly.

Next we might want to check that a parameter contains values we expect:

```
%if %upcase(&RequiredParm1.) ne Y and %upcase(&RequiredParm1.) ne N %then %do;
    %put ERROR: RequiredParm1 must be either Y or N;
    %put        RequiredParm1 = **&RequiredParm1 .**;
    %let __Macro_Err=1;
%end;
```

Note that this check will allow both upper case and lower case values, which may or may not be appropriate for your checking regimen. When I provide an error message that explains what the user passed to the macro, I surround the result in asterisks. This lets the user see any leading or trailing spaces stored within, and thus get a much better handle on what is actually stored in the macro variable you are displaying.

We could also look at the value of one parameter relative to that of another parameter for example if `&RequiredParm1.` is "N" then we might want to make sure "`&RequiredParm2.` is also "N" so that would look something like:

```
%if %upcase(&RequiredParm1.) eq N and %upcase(&RequiredParm2.) ne N %then %do;
    %put ERROR: RequiredParm2 must be N when RequiredParm1 is N!;
    %put        RequiredParm1 = **&RequiredParm1.**;
    %put        RequiredParm2 = **&RequiredParm2.**;
    %let __Macro_Err=1;
%end;
```

Or if we are looking at a power or sample size calculation we might check something like this:

```
%if (&AlphaLevel. GT 1) or (&AlphaLevel. LT 0) %then %do;
    %put ERROR: AlphaLevel is expected to be between 0.0 and 1.0;
    %put        AlphaLevel = **&AlphaLevel.**;
    %let __Macro_Err=1;
%end;
%if (&BetaLevel. GT 1) or (&BetaLevel. LT 0) %then %do;
    %put ERROR: BetaLevel is expected to be between 0.0 and 1.0;
    %put        BetaLevel = **&BetaLevel.**;
    %let __Macro_Err=1;
%end;
```

in order to ensure that the α– and β–levels are in the expected ranges.

Some error checks like these are trivial, was the variable even provided or did the correct value get entered. Simple if-then clauses are sufficient for these sorts of things. However using this methodology, more interesting and difficult checks can be preformed. For example this segment of code checks to make sure the dataset is there before we ever encounter a SAS® error:

```
%local __DS_Exists;
%let __DS_Exists=1;
%if %sysfunc(exist(&ds.)) eq 0 %then %do
    %put ERROR: The dataset named &ds. does not exist;
    %let __Macro_Err=1;
    %let __DS_Exists=0;
%end;
```

By using the `%SYSFUNC()` macro function, the SAS® function `EXIST()` becomes a macro function and tells us if `&ds.` exists. Now we can perform data specific checks on `&ds.` We should first check that `&ds.` isn't empty, but if the user provided a name for `&ds.` and a SAS® error did not occur trying to access it. The `EXIST()` function tells us the database's status We also in the process make another macro flag to let us know that our dataset exists – a pretty useful bit of information to know.

Think that's slick? Now for example we can only perform some parameter checks if we know we have the correct dataset specified. How about we make sure the provided variable is even in the dataset listed provided to the macro (`&ds.`). Note we only run this bit if `&__DS_Exists` is true:

```
%if &__DS_Exists eq 1 %then %do;
    proc sql noprint;
        select name into:__AllVarsin_DS separated by " "
        from sashelp.vcolumn
        where libname='WORK' & memname=%upcase("&ds.");
        quit;
    run;
    %if %index(%upcase(&__AllVarsin_DS.),%upcase(&RequiredParm1.)) eq 0 %then %do;
        %put ERROR: &RequiredParm1. is not found in &ds..;
        %let __Macro_Err=1;
    %end;
%end;
```

It does this by first getting a list from the `SASHELP.VCOLUMN` dataset which is created by default and using the `SQL` procedure to get a macro variable list separated by a blank space of all the variables in the dataset. We then use `%INDEX()` to check to ensure the variable provided is actually IN the `&ds.` dataset. I can not begin to tell you how many hours these 10 lines of code have saved me in later debugging when someone just mistyped "Gender" as "Gedner". This works well for smaller databases, but larger datasets with long lists of variables might need to use a

different construct (e.g. the `&&Var&i.` structure talked about in Carpenter's Complete Guide to the SAS® Macro Language), but the basic idea is the same:

```
%if &__DS_Exists eq 1 %then %do;
    proc sql noprint;
        select name into:__Vars1-:Vars9999
        from sashelp.vcolumn
        where libname='WORK' & memname=%upcase("&ds.");
        quit;
    run;
    %let VarCount = &SqlObs;

    %let VarFound = 0;
    %do _Counter_ = 1 %to &VarCount.;
    %if %upcase(&&Vars&_Counter_.) eq %upcase(&RequiredParm1.) %then
        %let VarFound = 1;
    %end;
    %if &VarFound. eq 0 %then %do;
        %put ERROR: &RequiredParm1. is not found in &ds..;
        %let __Macro_Err=1;
    %end;
%end;
```

The sky is really the limit to what can be checked against. As long as you follow the basic pattern of:

```
%if <<CONDITION(S)>> %then %do
    %put ERROR: <<ERROR MESSAGE>>;
    %let __Macro_Err=1;
%end;
```

the macro can pretty well check any condition that you can conceive of.

GETTING OUT

So rather than stopping the macro at each little error, I like to check all the parameters at once and figure out which ones have problems. Once we've checked them all for errors, if there's a problem, we stop the macro using:

```
%if &__Macro_Err. %then %do;
    data _null_;
       abort 3;
    run;
%end;
```

This stops processing and sends an error code different from the standard SAS® exit code, so we know it was a problem in the macro as opposed to other areas of the program. This lets the savvy user know better where to look for trouble in a longer code block. Further if there are multiple places where this macro could stop and abort – a unique code parameter problem code (I tend to use 3) will inform us that the problem was with the parameter checking rather than another code block.

TRICK #2: THE &DEBUG. PARAMETER

Another coding trick I use in development of such proc replacement type macros is to use formal "DEBUG" parameters in my code. I then use the macro variable `&DEBUG.` as a Boolean variable to tell me if I should put out the debugging information. Once development is finished and this moves to production, these serve as invaluable aids for discovery of troubles when the users inevitably break code with data that I never thought would hit the program. For example this section:

```
%if &DEBUG eq 1 %then %do;
    %let DEBUGCount = %trim(%left(%sysevalf(&DebugCount. + 1)));
    %put ************************************************;
    %put DEBUG Point &DebugCount ;
    proc print data = ReadersOut;
        var &IDvar.
            %do Popem = 1 %TO &NumObsCuts.;
                Sens&PopEm. FPR&PopEm.
            %end;;
    run;
    %put ************************************************;
%end;
```

runs a `PRINT` procedure of the `ReadersOut` database. This ONLY occurs if the debugging is turned on, so it never even gets called if `DEBUG` is off. In this example, I use a macro variable to count which debug point is being produced to make them easier to find in the log. By initializing a local macro variable called `&DebugCount.` to zero at the start of any given macro, and adding in the second line, I never have to keep a count in my code, I just run the code and it tells me it's the "Xth" debug point. Useful in tracking down errors later, when plodding through log files that call the macro from a library or a `%INCLUDE` statement. However, since I originally developed this technique I have taken to using mnemonic markers rather than a debug count. Again for searching ease, as it is much simpler to do a search for "`Sensitivity and FPR Pop Check`" than to remember that such a point is `DEBUG Point #3` months after the code is originally written.

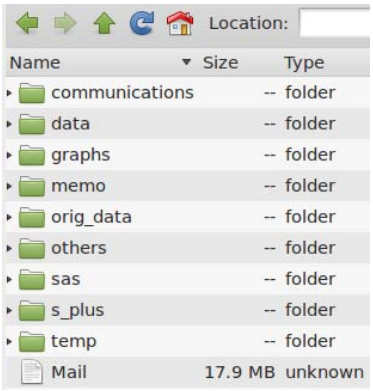
TRICK #3: USING MACRO VARIABLES A "CONSTANTS"

This is another trick I use which many have discussed. Other programming languages like C, Perl or Java have the ability to define constants that are used through out the program. This is useful for saving repeatedly used things like the program's base path, or to store a common conversion factors like the company's accepted number of digits for PI, or the number of pounds in a kilogram. These mnemonic reminders make reading the code later much easier, and allow changes to be made in only one place. Ideally they might even be stored in an `%include` file in a common library area, depending on their nature and use within the company

When putting them directly in a specific program, my preference is to place all of these at the beginning of my code, offset by a comment block:

```
/*********************v*************************************************
 Defining Constants
 ********************************************************************************/;
%let BASEDIR = /proj/alphabeta/Projects/Nursing/Smith/Proj12345;
%let LOCALINCLUDEDIR =  &BASEDIR./sas/_include;
%let GLOBALINCLUDEDIR = /home/karafam/GlobalIncludes;
%let PI = 3.14159265
%let KG2LBS = 2.20462262; * 1 KG = ~2.2LBS;
%let DAYSPERYEAR = 365.2524; ** Accounts for leap seconds IIRC, I have OCD clients;
/*********************************************************************************/;
```

Comments about the constants use (like KG2LBS)These constants can be used in open code, or with in macros once defined, as they reside in the global macro table. To keep them more distinct from other macro variables I tend to use the stylistic trick of all caps for these kinds of constants. The assumption is that with in a given program they shouldn't change.



TRICK #4: USE STRUCTURED DIRECTORY LAYOUT

This tip, doesn't necessarily involve SAS® directly, but how one lays out their specific project structure. Knowing what sort of directory structure to expect for a given project allows one to code to a common project object. For example, in my department our working group uses the common directory layout depicted at the right. We create separate folders to store specific things like working vs. original data. For any given project you can write code to assume this structure

and you know your data is always stored in `&BASEDIR./data` if it's working data or `&BASEDIR/orig_data` if it's the original data from the data source. Then all that code needs is a macro constant specifying the base directory or folder for the project to point to the data.

An area is created for things that come from the client (./communications), and things that go to the client (./memo). E-mail for the project is stored within the project folder as well. This is useful when projects are archived later as everything about the study is stored in one location. Statistical programs types have their own folders, so that all SAS® files can be found together and assumed to be in the ./sas directory, while R and SPLUS files are in the ./s_plus folder. Our group has gone so far as to have a series of project creation scripts that set this layout up, link it in to our IMAP mailbox storage and file it appropriately if multiple users are sharing a common folder.

The biggest benefit we have found from such a structure is project hand off. When someone inevitably leaves or is replaced on a project and a new analyst has to pick up the reigns, things are stored in a common way and the methods for revising the code are more clear based on the common directory structure. Also when several statisticians and programmers are working on the same study, the layout again facilitates communication and documentation more easily.

TRICK #5: DEVELOP A "WORKHORSE" LIBRARY

Many of us have developed a list of "proc replacement" macros. These either completely revise output or handling of how a given procedure does something, or generate a specific plot or table. These are not the "workhorse" macros I'm referring to here. A work horse macro is likely a small 5-10 line macro that does one specific function that you will use over and over in many macros. For example the macro listed below:

```
%macro cv(x);
    %if length(&x.) eq 0 %then %do;
       %put ERROR: X is a required macro in the %cv() macro!;
       %put        Exiting due to %cv() failure;
       data _null_;
           abort 3;
       run;
    %end;
    %let p=0;
    %do %while(%scan(&x,&p+1) ^=    );
        %let p=%eval(&p+1);
    %end;
    &p.
%mend;
```

This little workhorse counts the number of items in a macro list and returns the number. Thus in some other macro I can call:

```
%let VariablesToProcess = %cv(&InputList.);
```

When we resolve `%cv()`, it returns the number of items in `&InputList.` that are separated by the standard set of delimiters – something I find myself doing all the time in macros. The advantage of having it in one place is I can reference it using `%include` and then only write the code for `%cv()` ***ONCE***. Meaning anytime I find I want to update the macro I change it in one place and all of the uses of `%cv()` will be updated instantly. If a new version of SAS® is released or as a result of something I pick up from a conference I want to improve on the method I use to count variables, I can change it in one place and all of my macros which rely on that code are improved. So for a macro that was popping off variables in a list for a `proc freq` call we might use this methodology:

```
%macro FreqishExample(VariableList = , ByVariable = );
     %*************************************************************************;
     %* Simple macro to perform multiple frequency tables against a single ;
     %* variable;
     %*************************************************************************;

     %*************************************************************************;
     %* Inlcuded Macros: Count Var;
     %*************************************************************************;
     %include "cv.sas"; %* Note: You will need a full path to cv.sas;

     %*************************************************************************;
     %* Parameter Checking Section;
     %*************************************************************************;
     %local __Macro_Err;
     %let   __Macro_Err = 0;
     %if length(&VariableList.) eq 0 %then %do;
       %put ERROR: &VariableList. needs at least 1 variable;
     %end;
     %if length(&ByVariable.) eq 0 %then %do;
       %put ERROR: &ByVariable. is expecting a variable name;
     %end;
     %if %cv(&ByVariable.) gt 1 %then %do;
       %put ERROR: &ByVariable. is expecting a SINGLE variable name;
     %end;
     %if __Macro_Err = 1 %then %do;
        data _null_;
           abort 3;
        run;
     %end;

     %*************************************************************************;
     %* Main Macro;
     %*************************************************************************;
     %let NumVars = %cv(&VariableList.);
     %do _I_ = 1 %to &NumVars;
         proc Freq;
             table &ByVariable. * %scan(&VariableList.,&_I_.);
             title "Table &_I_.: &ByVariable. * %scan(&VariableList.,&_I_.)";
         run;
     %end;
%mend; %*Ending FreqishExample;
```

While this is a simple example it comprises many of the tips used. The idea of modular programming is far from new, its how most of the software developed is done. Break the task in to smaller tasks, use and reuse modules as much as possible. Things to worry about here are that changes to parameter requirements and return values will break functionality. BUT maintained correctly it is an extremely powerful tool for you macro toolbox.

CONCLUSION

This paper describes a programming technique that can easily be fitted to library level macros for parameter checking and several other tips that can make your macro coding life more streamlined and professional looking. These types of checks not only cut down on unnecessary debugging when conditions are not properly met, but they also provide the macro a more polished feel and a more professional completeness that takes them from the solo macro to a library grade program.

ACKNOWLEDGMENTS

The author would like to thank Ralph O'Brien and Art Carpenter whose mentorship and teaching in coding practice directly lead to most of the ideas presented here.

RECOMMENDED READING

The Next Step: Integrating the Software Development Life Cycle with SAS® Programming, Gill

SAS® Macro Programming Made Easy, Burlew

Carpenter's Complete Guide to the SAS® Macro Language, Carpenter

CONTACT INFORMATION

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

Name: Matthew T. Karafa, PhD
Enterprise: Quantitative Health Sciences, Cleveland Clinic Foundation
Address: 9500 Euclid Avenue / JJN3-01
City, State ZIP: Cleveland, Ohio 44195
Work Phone: (216) 445-9556
Fax:
E-mail: karafam@ccf.org
Web: