Paper TS02

Building Better Macros: Basic Parameter Checking for Avoiding "ID10T" 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 my parameter checking scheme, which provides such users direct feedback via the SAS® log. This feedback looks like a standard SAS error message, and it gives meaningful explanations of what went wrong. 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. These sorts of macros, that are designed to be out in the wild, NEED parameter checking to make sure this kind of scenario doesn't happen.

PROGRAMMING STRATEGY

The basic strategy starts with the assumption that the user has entered all the parameters correctly. 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) Set the error flag to true; 2) provide an error message using `%PUT`. 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 we 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've 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. 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 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 explain what the user passed to the macro, I surround the result in asterisks. This lets the user see any leading or trailing spaces and get a much better handle on what is actually stored in the macro variable you are displaying.

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.` if the user provided it and not incur a SAS® error mentioning the lack of `&ds.` if we mistyped it. 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? 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”. 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.

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

```
%if <<CONDITION>> %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.

ADDITIONAL TRICKS

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. 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. Very useful in tracking down errors later, when plodding through log files that call the macro from a library or a `%INCLUDE` statement.

CONCLUSION

This paper describes a programming technique that can easily be fitted to library level macros for parameter checking. 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

Carpener’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: