

A Second Look at SAS® Macro Design Issues

Ian Whitlock, Kennett Square, PA

ABSTRACT

At SUGI 27 the author presented a paper "SAS Macro Design Issues" as a Beginning Tutorial. The use of names and parameters were emphasized because it was presented as a Beginning Tutorial.

This tutorial assumes that you have written some macros, made macro decisions with %IF and done looping with %DO statements, but you are still teachable, *i.e.* bad habits are not so deeply ingrained that there is nothing more you can learn about macro.

Two questions motivated this paper. The first came more than twenty years ago at the end of my first programming course when the professor asked, "You now know how to write programs, but do you know when it is appropriate to write a program?" For SAS macro the question might be rephrased. What kind of SAS problem is appropriate to macro? Or when should the solution involve macro code? The second came from SAS-L in September 2001, "What sources discuss how to develop clear and good macro code?"

The material discussed was developed under Windows, but is appropriate to all operating systems.

INTRODUCTION

The first point to understanding how to design and write macros is to understand that macro is not a programming language for manipulating data. It is a programming language, but the product, with the exception of side effects, is SAS code. Consequently the question to always keep in mind is - What **code** do you want to produce?

Here are a set of problems taken from SAS-L:

- 1) Generate random variable creating a number of datasets.
- 2) Modify a master file based on transactions where the transaction sets may be empty.
- 3) Make sets of global macro variables holding lists of dataset variable names.
- 4) Report parameter values.

I will look at these problems from the point of view of the person raising the question to illustrate some macro misconceptions and readability issues. Then I will illustrate design principles with an emphasis on list processing, utilities, and macro communication in my solutions.

DATA STEP CODE VERSUS MACRO CODE

The example, shows the difference between DATA step code and the corresponding macro code, and emphasizes the role of parameters in building flexible and readable code.

Allison, new to macro, wanted to generate random variables X and Y in a number of DATA sets. Her code:

```
%macro DOIT;
  data test1 test2 test3;
    %do j=1 %to 10;
      x=ranuni(0);
      y=mod(int(10*x),3);

      %if y=0 %then output test3;
      %else %if y=1 %then output test1;
      %else %if y=2 %then output test2;
    %end;
run;

%do i=1 %to 3;
  proc print data=test&i;
    title2 "test&i";
```

```

        run;
    %end;
%mend DOIT;

%DOIT;

```

Her question - why is only one record generated for each dataset. Remember to ask what code is generated. For each iteration of the first %DO-loop the code

```

x=ranuni(0);
y=mod(int(10*x),3);

```

is generated. What happened to the OUTPUT statements? Well the %IF tests all fail because the letter "y" is not a number. Remember that macro executes at compile time. The variable Y has not yet been created; macro works on text. Here we want to decide whether to execute an OUTPUT statement or not. Consequently a DATA step IF is required. Use %IF when you want to decide whether to generate the code. Use IF to decide whether you want to execute the written code or not.

Although Allison was not really interested in design issues, the example is worth looking at from this point of view. First I note that %DOIT has no parameters and is a poorly chosen name. The name is poorly chosen because the calling code

```
%DOIT;
```

gives no indication of what is going to be done. The lack of parameters almost always indicates either that there are macro variables which should be parameters, or that the macro is too specifically written. In other words, there is constant SAS code in the macro that should really be a variable. In this case the prime candidates are:

- 1) How many datasets should be produced?
- 2) What is the common root of the names?
- 3) How many observations should be distributed?
- 4) What variable name should the random variable, X, have?
- 5) What seed should be used in the random number generator?
- 6) What distribution function should be used to generate the random numbers?
- 7) Are test prints desired?

You should be able to think of more possible parameters.

Should the variable Y be on the output files? It is really used to decide where to send the observation. Consequently, I decided to drop it.

The original code has %DO-loop. Should it be a DATA step loop or a macro one? As a %DO the assignment and output statements are generated many times. The whole purpose of DATA step DO-loops is to reduce the amount of code compiled. Hence a DATA step loop is better here. In contrast, the loop generating the test prints must be a macro loop.

The decision to use a DATA step DO-loop to loop over the assignment and SELECT block means another variable that should probably not be on the output datasets. Now we have two DATA variables Y and J that prevent the consumer of the macro from using these names for his random variable. In general you will have to introduce DATA variables in your macros for the macro's purpose and not because the consumer has any interest in these variables; hence they should be dropped. Moreover, they should not restrict the consumer from having certain names. I use the common rule that variables beginning with a double underscore belong to the macro developer and should not be used by the consumer. In any case such variables should not be written out to any consumer dataset.

In terms of a real problem, you should ask whether the output should be one dataset or multiple sets. Here I have kept with Allison's intent. However, in my SAS-L posting I had a parameter to let the user decide whether the output should be one or multiple files.

Here is the macro.

```

%macro Random_data
( outroot = test /* root for output data sets */
, nobs = 10 /* num obs in total */
, nsets = 3 /* determines split var */
, var = x /* name of the random variable */
, seed = 0 /* seed for random number generation */

```

```

, debug = no      /* are prints wanted?          */
) ;
/* -----
   Generate uniform random number variable in multiple
   datasets with names having a common root
   -----
*/
%local i          /* looping variable          */
      dslist      /* list of generated datasets */
;
%let debug = %upcase ( &debug ) ;

%do i = 1 %to &nsets ;
    %let dslist = &dslist &outroot&i ;
%end ;

data &dslist ;
    drop __j __y;
    do __j=1 to &nobs;
        &var=ranuni(&seed);
        __y=mod(int(10*&var),&nsets);

        select ( __y ) ;
            %do i = 0 %to &nsets - 1;
                when (&i) output &outroot%eval(&i+1) ;
            %end ;
            otherwise error ;
        end ;
    end ;
run;

%if &debug = YES %then
%do ;
    %do i=1 %to &nsets ;
        proc print data=&outroot&i;
            title2 "&outroot&i";
        run;
    %end ;
%end;
%mend random_data ;

```

First look at the macro header. It is a good idea to add a short comment about each parameter. In addition, key word parameters have been required to enhance readability of the calling code. Also note that the parameter names have been chosen to make the calling code more readable. In general, I believe that key word parameters should almost always be used for the above reasons. The one exception that I usually allow is for macros that are designed to act like DATA step functions.

I placed each parameter on a separate line so that there would be room for a simple comment, and placed the separating comma in front of each parameter instead of behind. This means that the lines can be moved around more easily and you are less likely to make mistakes in adding new parameters.

Immediately after the macro header, I placed a short comment block describing the purpose of the macro. In a more realistic setting you would have more information about the macro in this comment block.

Note the use of %LOCAL to declare all variables that are not parameters. I will discuss the issues of global macro variables in a later section.

The macro variable DSLIST was used to allow separation of the macro code from the DATA step statement. You often find code like

```

data %do i = 1 %to &nsets ;
    &outroot&i
    %end ;
;

```

This code is correct, but harder to read because the macro code has been intertwined with the SAS code. In general it is better to separate the macro instructions from the SAS code. In another section I will use a macro to achieve a similar purpose.

The SELECT block requires a macro loop. Why? Because we want to generate a sequence of WHEN statements. This could not be done with a DATA step loop. Why do we need a loop at all? Because SAS does not allow an OUTPUT statement to be controlled by a data variable. Often macro code is used to simply enhance or make the SAS language appear more dynamic than it is. A lot of the time this means generating IF/ELSE statements or SELECT blocks to describe the dynamic situation.

Here is the test code.

```
%Random_data ( outroot = w, nob = 5, nsets = 2, var = z, debug = yes )

%random_data ()
```

It is always a good idea to test with default values and to test without default values. In general, testing macros requires much more work than testing SAS code precisely because macros should provide for many more possible situations through the use of parameters.

The parameter list is extremely important for readability because it describes the contract between the macro and the consumer. It essentially says, if these values are provided then the macro will provide the code to perform the described service. Any values not explicitly provided by the consumer will default to the value provided in the macro header. This means that the reader of the calling code can know exactly what is to be done by looking at the macro call and possibly the macro header. It is also extremely important because it allows the consumer to get varied services without needing to mess with the macro code.

GET THE TASKS RIGHT

The example illustrates what happens when a macro is too specific, i.e. has too much knowledge about the problem. It also provides an opportunity to look at the role of utility macros, and a comparison of controlling the flow in a managing macro versus controlling it from within.

Bart had a transaction processing problem. He had a master file and three transaction files. Whenever a transaction file was not empty she wanted to have those transactions applied to the master file via the MODIFY statement. Here is his code.

```
%macro numobs1(dsn);
  %global dsid num1 rc;
  %let num1=0;
  %let dsid=0;
  %let rc=0;

  data _null_;
    %let dsid=%sysfunc(open(&dsn));
    %let num1=%sysfunc(attrn(&dsid,nobs));
    %let rc=%sysfunc(close(&dsid));
  %mend numobs1;

%macro numobs2(dsn);
  %global dsid num2 rc;
  %let num2=0;
  %let dsid=0;
  %let rc=0;

  data _null_;
    %let dsid=%sysfunc(open(&dsn));
    %let num2=%sysfunc(attrn(&dsid,nobs));
    %let rc=%sysfunc(close(&dsid));
  %mend numobs2;

%macro numobs3(dsn);
  %global dsid num3 rc;
```

```

%let num3=0;
%let dsid=0;
%let rc=0;

data _null_;
%let dsid=%sysfunc(open(&dsn));
%let num3=%sysfunc(attrn(&dsid,nobs));
%let rc=%sysfunc(close(&dsid));
%mend numobs3;

%macro moddsn;
  %if &num1 > 0 %then %do;
    data work.modprod;
      modify work.modprod work.modprod2;
      by pd st md;
    %end;

    %if &num2 > 0 %then %do;
      data work.modprod;
        modify work.modprod work.modprod3;
        by pd st md;
      %end;

      %if &num3 > 0 %then %do;
        data work.modprod;
          modify work.modprod work.modprod4;
          by pd st md;
        %end;
      %end;
    %end;
  %end;
%mend moddsn;

```

The first thing that should strike you is the repetitive quality of the three NUMOBS macros. The only essential difference is that each creates a different global variable. That means a parameter is missing. The macro needs to be given the name of the global macro variable rather than have it built in.

On a more detailed level you should wonder why RC and DSID are declared global, since there is no need to know the values of these variables outside the macro. They should be declared local. Now what about NUM1 NUM2 NUM3? These variables must be known in the macro %MODDSN, but the values are assigned in submacros; hence Bart made them global. This is a common technique for communicating between macros.

On the other hand, only %MODDSN has to use this value; hence they really should be owned by %MODDSN and assigned by the corresponding submacro. That brings up the question of where the macros %NUMOBS1, 2, and 3 are called. They were called independently before %MODDSN. That is really a defect in the design of the solution. %MODDSN wants the results, so it really should make the calls, because then the relationship between the macros is shown. We do not have 4 independent macros. We really have a main macro and three submacros to provide the corresponding number of transactions. The code should reflect that fact rather than leave it to the reader to discover.

Again, why were three variables needed instead of one? Because the calls were made outside %MODDSN, three different variables were needed to hold the three different values. By placing the calls inside %MODDSN we can use one variable three times instead of requiring three variables. Since %MODDSN makes the calls, it can also be responsible for declaring the variable local; hence no global macro variables are needed at all. Or to turn it around, the required use of global variables indicated defects in the communication between the macros because of defects in the design process. This provides a very good reason to question the design of any program that requires the use of global macro variables.

In general, global macro variables are a poor choice of communication technique, not only because there are better solutions, but also because they leave you open to having any macro possibly have the ability to change their values. This means that you as the programmer have to know about the whole system at once in order to know how it works. Consequently the size

and complexity of the program that you can write are severely limited, and the interaction between programmers working on a common system becomes a major problem.

The one legitimate use of global macro variables is when they are really used as parameters for the entire system and it is agreed that no part of the system will change them once they have been assigned. Unfortunately, SAS leaves the burden to the programmer, rather than enforcing this rule through the language.

Using the above considerations we can reduce the three NUMOBS macros to one.

```
%macro numobs(dsn);
  %local dsid rc;
  %let num=0;
  %let dsid=0;
  %let rc=0;

  data _null_;
  %let dsid=%sysfunc(open(&dsn));
  %let num=%sysfunc(attrn(&dsid,nobs));
  %let rc=%sysfunc(close(&dsid));
%mend numobs;
```

There are still several problems here. NUM is assigned in NUMOBS, but it must be declared in the calling macro, %MODDSN. Unfortunately, as written, SAS macro requires this. On the other hand, it is not too bad to ask the owner of the variable to declare it. There is no reason for the DATA_NULL_ step. Using the DATA step functions with %SYSFUNC to execute them, the code could be pure macro code, which means it could in principle be called in a macro instruction.

Now consider:

```
%macro numobs
  (data=&syslast /* true SAS dataset required */
  ) ;
  /* -----
   Return number of obs as a function
   -----
  */
  %local dsid nobs rc;
  %let data = &data ; /* force evaluation of &SYSLAST */
  %let dsid=%sysfunc(open(&data));
  %if &dsid > 0 %then
  %do ;
    %let nobs=%sysfunc(attrn(&dsid,nobs));
    %let rc=%sysfunc(close(&dsid));
  %end ;
  %else
    %let nobs = -1 ;
  &nobs
  %mend numobs;
```

This macro could be called with

```
%let myvar = %numobs(data=test) ;
```

In other words, %NUMOBS no longer needs to assign the variable because it generates the value with true macro instructions. Hence, it does not need to know the name of the variable and the function of the macro is now truly self contained.

Consequently there are no secret agreements between %MODDSN and %NUMOBS. This makes %NUMOBS easier to use and the calling code easier to read by spelling out the service provided in the code.

I have made several other improvements. The value -1 is used to return errors. I have left it to the caller to handle the error, since %NUMOBS is not really in a position to know how to handle the problem. You could modify the code to report the error to the log, but this is liable to produce multiple messages if the caller cannot handle the error. The parameter DATA has been given the standard SAS default to use the last created SAS dataset. However, it is important to note that the parameter value is &SYSLAST, *i.e.* a macro instruction to resolve a variable; it is not the value of that variable. In such cases it is best to force the evaluation on the first executable line with:

```
%let data = &data ; /* force evaluation of &SYSLAST */
```

It is a good idea to add the comment to prevent someone from deciding that the line is not needed and removing it.

You should also be aware of the limitations of the macro. The ATTRN function only works with true native SAS datasets. It cannot handle SAS views or data read through engines to foreign databases. Even with SAS data, the call does not work when observations have been removed, but not physically deleted. This last restriction could easily be removed by use LOBS instead of NOBS with the ATTRN function.

Now it is time to look at %MODDSN. As given above, too much information about the specific problem is built into the macro. The master file must be called WORK.MODPROD and the transaction files must be called WORK.MODPROD2, 3, and 4. By now you should recognize the problem as a failure to consider appropriate parameters. Should MODDSN work with three transaction files, or just one? If it worked with just one then it would be more useful, and handling three transaction files would simply mean that three calls to macro are needed instead of one. In general, you should learn to distinguish between details of your specific problem and the utility task. In this case the utility is to protect the MODIFY DATA step from an empty transaction file, and the project specific work requires three oddly named transaction files. For example, the variable names PD, ST, and ME are clearly usage specific and have no business in general code. You know that means a missing parameter.

These considerations lead to the code:

```
%macro moddsn
  (master=work.modprod /* SAS dataset to update */
  ,trans=work.modprod2 /* transaction to be applied */
  ,key=pd st me /* record level key to both datasets */
  );
  /* -----
  apply transactions to master file when not empty
  ----- */
  %if %numobs(data=&master) > 0 %then
  %do;
    data &master;
      modify &master &trans;
      by &key;
    run ;
  %end;
%end ;
%mend moddsn;
```

It is interesting to note that careful attention to design considerations have led to shorter, simpler, easier to use macros with more readable calling code.

ANOTHER GLOBAL VARIABLE PROBLEM

Connor wants to simplify running many PROC MEANS on a SAS dataset, TEMPA, with approximately 1200 variables. He has created a control dataset, TEMPB, that knows the names of the variables and has three special attributes of the variables called HIT, FUNCTION, and OBJECT. The variable names are held in a field called PROGID2.

He has a macro VARLIST which uses TEMP that should construct the name of a global macro variable and assign it a list of variable names from TEMPA. Example usage code might be:

```
%Varlist(TempB, Function,120)
Proc Means Data = TempA N NMiss Sum ;
```

```

        Var &Function120 ;
run ;

```

Here %VARLIST constructs the macro variable FUNTION120 by subsetting TEMPB to observations with FUNCTION=120 and assigning it the list of values in PROGID2 for those observations. You should recognize this as very similar in structure to the previous example, albeit very different detail.

His macro code:

```

%Macro Varlist(DSName,CodeType,CodeValue) ;
  %Global Listname ;
  %Let ListName = &CodeType&CodeValue ;
  proc sql noprint;
    select  ProgID2
      into  :&ListName separated by ' '
      from  &DSName
      where &CodeType EQ &CodeValue
    ;
  Quit ;
%MEnd Varlist ;

```

His communication problem was caused by failing to make the constructed name global. The simple fix would be:

```

%Macro Varlist(Control=tempb,CodeType=hit,CodeValue=2) ;
  %Global &CodeType&CodeValue ;
  proc sql noprint;
    select  ProgID2
      into  :&CodeType&CodeValue separated by ' '
      from  &DSName
      where &CodeType EQ &CodeValue
    ;
  Quit ;
%MEnd Varlist ;

```

However, he indicated that OBJECT has character values. This means that the where clause using CODEVALUE is in conflict with the macro variable name &CODETYPE&CODEVALUE, since one requires quote marks and the other cannot have quote marks. This could be solved with:

```

%Macro Varlist(Control=tempb,CodeType=hit,CodeValue=2) ;
  %Global &CodeType&CodeValue ;
  %local qCodeValue ;
  %if %upcase(&CodeValue) = OBJECT then
    %let qCodeValue = "&CodeValue" ;
  %else
    %let qCodeValue = &CodeValue ;
  proc sql noprint;
    select  ProgID2
      into  :&CodeType&CodeValue separated by ' '
      from  &DSName
      where &CodeType EQ &qCodeValue
    ;
  Quit ;
%Mend Varlist ;

```

This presents a workable solution, but does not solve the problem of global variables. Connor intends to generate a hundred global variables and then use them. I assume that the PROC MEANS simply provided an example usage rather than indicating that he wanted to run hundred different PROC MEANS.

What are the possibilities? I would probably rewrite %VARLIST using %SYSFUNC and DATA step functions to read the control dataset and create the list with pure macro code to obtain a solution similar to the one given for %NUMOBS. In the long run, his project description seems to warrant it. However, this would require some hard work, and such solutions are not always available, so let's look at some other possibilities.

We could say that there always should be some form of consuming macro. For example, continuing the illustration with PROC MEANS, say the macro is %RUNMEANS. Then the code might be:

```
%macro RunMeans
  (data=
  ,control=
  ,codetype=
  ,codevalue=
  ) ;
  /* -----
     Get list of variables from CONTROL and apply to DATA
     running PROC MEANS
     -----
  */
  %local &CodeType&CodeValue ;
  %varList(&control,&codetype,&codevalue)

  proc means data = &data N NMiss Sum ;
    Var &CodeType&CodeValue ;
  run ;
%mend RunMeans ;
```

And %VARLIST is as given above without the %GLOBAL statement. This solution makes %RUNMEANS the managing macro and %VARLIST the helping macro. Another possibility is to reverse the roles. For example, %GETLISTRUN gets the list of variables and then passes the list to a submacro to execute.

```
%macro GetListRun
  (control=tempb
  ,CodeType=hit
  ,CodeValue=2
  ,mac=means(data=tempa,vlist=list)) ;

  %local list ;
  proc sql noprint;
    select Progid2
      into :list separated by ' '
      from &DSName
      where &CodeType EQ &CodeValue
    ;
  Quit ;

  %&mac
%mend GetListRun ;

%macro means
  ( data=tempa
  , vlist=list
  ) ;

  proc means data = &data N NMiss Sum ;
```

```

        Var &&&vlist ;
run ;

%mend means ;

```

In this case, the parameter VLIST names a variable to hold the list; so &VLIST is the name of macro variable. Now two more ampersands are needed to reference that variable because the scanning process resolves two ampersands to one and at the end of the scan looks to see if any ampersands are present in the expression. If they are the expression is scanned again. So

```
&&&VLIST
```

first resolve to

```
&LIST
```

and then to the value of LIST, which is what is wanted.

Passing variable names instead of variable values is an important technique in communicating information from one macro to another, so it should be in your repertoire of methods to avoid global macro variables.

MORE TRIPLE AMPERSANDS

Let's reinforce the use of macro expressions involving triple ampersands using the request of Debra to have a macro for reporting parameters in the form:

```
>> & parametername = parametervalue <<
```

I would prefer the expression without the ampersand because I see it as causing confusion.

```
>> parametername = parametervalue <<
```

However, it is interesting to consider the problem with the ampersand in front of the name because it raises the question of macro quoting.

Since there is likely to be a long list of parameters when one wants such a macro, let's assume a list of parameter names instead of just one. Here is the code.

```

%macro parmrep
  (list /* provide a list of less than 999,999 macro variables */
  );
  /* -----
     Report macro variable names and value in the form
     >> &name = value <<
     -----
  */
  %local i w ;
  %do i = 1 %to 999999 ;
    %let w = %scan ( &list , &i ) ;
    %if %length(&w) = 0 %then %goto mexit ;
    %put >> %nrstr(&) &w = &&&w << ;
  %end ;
  %put ERROR: ParmRep LIST limited to 999999 variables ;
%mexit ;
%mend;

```

You can scan until an empty value is reached; however, this requires doing your own incrementation. In this case, I find it easier to assume a large upper bound and report an error if the bound is reached. I violated my general rule to use key word parameters because modification of the call to this macro is very unlikely and it has the feel of a positional type of macro. (Of course, this choice is questionable and very arbitrary.)

Each name is peeled off one at a time into the variable W. The length is tested and the macro ended when W is empty. If the end of the loop is actually executed then the code falls into an error message.

The macro quoting function %NRSTR is used to hide the ampersand from the macro facility. The ampersand in front of the W is needed in order to get the name of the macro variable. Now the triple ampersand is needed for the same reason given in the previous example.

CONCLUSION

You should now have a better idea of what macro is about, why you should use it, and how your code should be developed. It is worth noting that macro is hard to debug, hard to write, and hard to read. Thus it is worth learning good code habits to smooth the development process. The understanding of this paper should give you some of the tools needed to get started.

Your comments and questions are valued and encouraged. Contact the author (email preferred) at:

Author Name: Ian Whitlock
Address: 29 Lonsdale Lane
City State ZIP: Kennett Square, PA 19348
Work Phone: 610-388-4358
Email: ian.whitlock@comcast.net

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.