

SAS® Macro Design Issues

Ian Whitlock, Kennett Square PA

Abstract

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?" There have been many Beginning Tutorials on macro, and many of the examples in them may illustrate some of the principles that these questions hint at, but I do not know of any source that concentrates on the principles.

To sum up, I will consider: 1) What makes a macro good? and 2) How do you make a maintainable, useful, clear system of macros? Do you need to know how to write %IF and %DO statements to understand this talk? No, all terms will be introduced, however, knowing something about these statements will increase your appreciation of the subject.

Introduction

First we will look at macro variables, see how to define and use them in a SAS program, and then consider what distinguishes a good variable from a poor one. In short, we will look at the parameterization of a SAS program.

Next we will consider what a macro is, look at how to define simple macros with parameters, and see how to and use them. We will explore why this is a more powerful concept than the parameterization of a SAS program.

Macro Variables

A macro variable can be created with a %LET statement.

```
%let data = lib.mydata ;
```

The variable is called DATA and has the value LIB.MYDATA. From the point of view of the macro facility, the value is just characters. We might think this is probably in a program where a libref, LIB, has been defined and that MYDATA refers to a member of this library. However, the macro facility knows none of this, and it is important to understand that the macro facility knows no SAS. It is simply a program operating on text data with embedded instructions that begin with a percent sign. As far as it is concerned, an instruction, %LET, is to be executed, creating a variable and assigning it a text value. Note that there are no quotes around the value. Why? All values are always text and it would be very annoying if everything had to be in quotes all the time.

In SAS, words have meaning, so it is necessary to distinguish character values with quotes. In macro it is the other way around, everything is value and the instructions must be distinguished with special symbols. All macro instructions begin with a percent sign. So how can our variable be referenced? The letters, DATA, are just that (four letters), they are not the variable we just defined. To tell the macro facility we want the variable evaluated we need a special symbol, the ampersand, to indicate we want the value of the variable instead of just the word.

For example,

```
libname lib "c:\myproject\dat" ;  
  
%let data = lib.mydata ;  
  
title3 "The data set is &data" ;  
proc print data = &data ;  
run ;
```

Even with this simple little example there is something significant going on. We referenced the variable twice. That means that in both places we are insured that we will be referring to the same data set. This could be important. How often have you seen a print out of synch with its title? That cannot happen here.

What makes this a good example? Well the name is chosen in a meaningful fashion so that any SAS programmer will probably recognize it as specifying a SAS data set. Second, the value is a complete unit, consequently little is lost in terms of readability.

To see the difference, let's consider a problem with character dates. Suppose the dates are in the form - three character month, space, day, space, year (e.g. DATE = "Jan 10 2002") and you want to convert to SAS date. If it had been DATE = "10Jan2002" then there would have been no problem with a solution:

```
sasdate = input ( date , date9. );
```

But in the given form, the pieces must be rearranged. One programmer considered:

```
scan(date,2)||scan(date,1)||scan(date,3)
```

After considering the expression and seeing the repetition of "SCAN(DATE," he decided to make this piece a macro variable. Can you do that? Yes, of course, remember the macro facility sees all values as plain text, so why should it care? The programmer wrote:

```
%let x = scan(date, ;  
sdt=input(&x 2)||&x 1)||&x 3),date9.);
```

The code works, it is legal, the programmer followed a good principle - make a macro variable to capture the repetition; but the code is a monstrosity. Why? What went wrong?

It is a monstrosity because the second line is impossible to read, and looks downright wrong. There is one left parenthesis and 4 right parentheses! The only thing that makes it at all possible to understand is the immediate proximity of the assignment to X. The variable name is very poorly chosen, but the assigned value is so poor that no name could help. The problem here is that the value cannot act as a unit, hence the code written has no readable interpretation. Thus we find that not all possible values for macro variables are reasonable values.

The syntax of SAS places some restrictions on the programmer as to what code can be written. The macro facility places far less restrictions on the programmer because it knows no SAS. Consequently it is far easier to write really unreadable code that is legal and will execute correctly.

Now, let's return to the PROC PRINT example at the beginning of this section. A second important principle is illustrated. We lifted a piece of code, the data set name, out of context and were able to control this piece of code from another place in the program, say at the top. In this sense one usually refers to the variable as a parameter of the program. The ability to define macro variables gives one the ability move where a complex code expression appears in the code. This can make the code much easier to read and write, or it can make it much harder depending on the situation and the choices made.

In the example the variable, DATA, does not play a good role as a parameter because the amount of code is too small and its usage, to keep the title in sync with the PROC PRINT, is too limited for a typical program level problem. However, data sets often do make good candidates for program parameters. Imagine some complex sequence of analysis using data variables with standard names. Here one might easily wish to have a data set parameter so that the program could be run on many different data sets.

Also notice the title in this example is in double quotes. This is necessary for the macro facility to resolve the reference, &DATA. Your code is read by the word scanner, which breaks the code up into tokens for the compiler. Single quoted text is recognized as a single token, so the expression does not go to the macro facility. On the other hand, a double quote is a token itself, so the word scanner parses the text between double quotes into tokens and therefore will send the token, &DATA, to the macro facility for resolution before passing it to the compiler.

As another example consider a program that begins

```
filename in  
  "\\rk2\vol12302\stat\dat\ds1.txt" ;  
  
filename pgm  
  "\\rk2\vol12302\stat\progs" ;
```

```
libname stat
  "\\rk2\vol12302\stat\sasdat" ;
```

Here it is easy to imagine when the project is moved to another server or disk there is an opportunity to make mistakes. It might be helpful to have a parameters, say ROOT and DSNUM so that the code becomes

```
%let root = "\\rk2\vol12302\stat";
%let dsnum = 1 ;

filename in
  "&root\dat\ds&dsnum..txt";

filename pgm
  "&root\progs" ;

libname stat
  "&root\sasdat" ;
```

Now the changes needed to move the project are controlled in two macro variables and the code writing has actually become easier because an element of messy repetition has been captured in the macro variable, ROOT.

You should contrast this code with the assignment:

```
%let x = scan(date, ;
```

I claimed that assignment was very poorly chosen. In both cases the macro variables refer to a part of something. So being a part is not the problem. The variables, ROOT and DSNUM, refer to units or entities in spite of the fact that they are only parts. ROOT refers to a path, not the complete path, but never the less a path. Here, the fact that meaningful names can easily be chosen suggests that the values will be good ones.

Before leaving this example, you should note the double dot after &DSNUM. It is necessary to have two dots here. The question is - how can text immediately follow a reference? It is important that there be no spaces between the text and the reference, but the text is not part of the name. There must be some special symbol to say this is the end (and part) of the reference. The first dot ends the reference so a second dot is needed to separate the filename portion from the file extension.

No dot was placed after ROOT because the backward slash is a separator and ended the reference just as a space would. It would not have hurt to put a dot there, since it does end the reference. In general, it is better to only add the dot when necessary. In that way the reader does not have to continually ask - is that dot there because it has to be? Consider:

```
%let root = \\rk2\vol12302\stat\ ;

filename pgm
  "&root.progs" ;
```

While this code is technically correct, it is a lot less clear. You cannot know whether PROGS is a subdirectory or just the last part of the end directory on &ROOT. It also is misleading because the reader might mistake the dot for an extension separator.

Why are variables sometimes called variables and sometimes parameters? Parameters drive the whole program. In general a program does not change assigned parameters, except possibly to make the letters upper case for easy comparison. Variables, on the other hand, are assigned within the program to achieve some more immediate end, and are often reassigned when the objective has been achieved. Thus the choice of word tips off the reader to the intended usage of the variable.

Finally we should mention a new macro instruction. The %PUT statement writes a message on the log. To see the value of DATA we might write:

```
%put data=&data ;
```

Remember the characters "data=" are just that, characters to help understand what is on the log. There are some special forms.

```
%put _all_ ;
```

Writes all macro variables and their values to the log. Note the word "_ALL_" must be by itself. If there is anything other than just spaces then it just means a sequence of characters.

```
%put _user_ ;
```

Writes all the macro variables that you create to the log. There are still other special forms, but you can look them up under %PUT.

Macros

For now, a macro can be thought of as a parameterized unit of SAS code. Let's reconsider our little problem of the print.

```
libname lib "c:\myproject\dat" ;  
  
%let data = lib.mydata ;  
  
title3 "The data set is &data" ;  
proc print data = &data ;  
run ;
```

A program typically might contain many prints and we might want to keep their respective data titles in sync the print that is actually made. Thus we want to think of the print as a unit of code. We define the macro as

```
%macro tprint (data=&syslast, tl=3) ;  
  
    title&tl "The data set is &data" ;  
    proc print data = &data ;  
    run ;  
    title&tl ;  
  
%mend tprint ;
```

We call the time when this code is read macro compile time. At this time the macro facility is storing away the information needed to generate our unit of code when needed. At this time no macro variables are resolved, rather the references are saved for resolution when we invoke the macro.

The name of the macro is TPRINT to suggest it's intended usage of make test prints of SAS data sets. The parameters are DATA and TL. We have used what are called keyword parameters because this form allows a default assignment and the consumer of the macro does not need to remember the order of the parameters. These two facts allow one to easily make changes in the macro without affecting any consumer code satisfied with what the macro current does.

You will often find others using positional parameters because they are traditional from the fact that the early programming languages used functions with positional parameters. When new parameters are added it becomes hard to remember their order, more code is needed in the macro to give the effect of default assignments and these defaults are not available to the consumer of the macro without documentation or the ability to read the macro code.

The first important design considerations in writing a macro are: 1) a good choice of the name to indicate its usage, 2) a good choice of parameters (and their names) to provide the consumer with options, and 3) good default values so that consumer often does not need to exercise those options. Then, of course, one must get the right code in the macro. It is far too easy to either put too much code or too little into a macro. You should be able to describe the task that the macro accomplishes in one sentence. If you cannot make the task description short then the task for the macro is not correctly defined.

The parameter name, DATA, was chosen to be consistent with SAS usage. Any SAS programmer expects a procedure to have a parameter DATA to specify the input SAS data set, thus it is easy to transfer this knowledge to the macro. The default value is a reference, &SYSLAST. SAS provides a number of macro variables automatically, which provide information about the system. SYSLAST provides the name of the last created data set. Since SAS typically uses the last created data set in a procedure when the data set is not specified, our macro should preserve this user expectation when reasonable. (Although I feel strongly that one should provide this default, I also feel, just as strongly, that no good SAS programmer should take advantage of this default. It is provided for SAS users, who do not consider themselves programmers.)

The parameter name, TL, stands for title line number. The first question should be - is the macro responsible for providing the title? Yes, that is the whole purpose of the macro, to provide a print with a synchronized data title. Ok, so now who should know the appropriate title line number? You cannot have a fixed line number because the consumer may have more title lines and it would be rather impolite for the macro to wipe out these lines. It is possible to write code to figure out the last used title line number and then make a variable TL to hold the next number, but that would make our macro more complex and detract

from its simple nature. As usual, when the macro cannot know a value or it is inconvenient to get it, then a parameter should be provided so that the consumer can provide it.

Now what about the default value. One would be a bad choice because this assumes the program has no titles. Ten would also be a poor choice because most programs use only a few title lines and the space to our title would be conspicuous. Three will not fit all people, but it will be good enough for most people most of the time because they rarely use more than two titles. If there is only one title the blank line will not hurt. Thus 3 is a good default value here.

Look at one more thing about the macro. At the end there is a line to remove the data title. Titles are global SAS statements and as such titles persist until changed or removed. Who should clean up? The macro consumer is not thinking about it and may not even be aware that a TITLEn statement is in the macro, so it would be wrong to ask her. The macro made the statement and the macro should be responsible for cleaning up after its usage is complete.

What about the LIBNAME statement? Should it be in the macro? No, remember we are providing a service for test prints. Usually the libref will already have been defined.

Now what about the RUN statement in the code? It is essential to allow the clean up of the data title. What if cleaning up the title were not an issue? It would still be necessary to allow the consumer to think of the macro as providing a completed task, in this case, making the print. In general, RUN statements are far more critical in good macro code than in SAS because macro instructions do not imply a step boundary. Thus without explicit step boundaries macro instruction may be executed at the wrong time.

So how do we use the macro? Macros are executed or invoked by placing a percent sign in front of the macro name and then providing the required list of parameters in parentheses. Thus the code

```
%tprint (data=test1)

%tprint (data=lib.mydata)
```

when executed makes two prints with two synchronized data titles that will not persist throughout the program containing these macro calls.

When the macro is invoked, the SAS code is generated. This time is called macro execution time. It is intertwined with the compiling (or parsing) of the generated SAS code. Finally there is the SAS execution time when the compiled code is executed. Thus there are four important times or stages through which the lines of a SAS program with macros move.

1. Macro compile time
2. Macro execution time
3. SAS compile time
4. SAS execution time

Typically these times are all intertwined and going on, at the same time for different lines (or parts of lines). It is important to distinguish because the different systems issue different kinds of error for different reasons, and because an understanding of these times helps you to understand what is happening in a SAS program involving macros.

Does a macro have to have parameters? No, and far too many macros are written without parameters. To illustrate consider the following code.

```
%macro mean ;

    proc means data = mydata ;
    run ;

%mend mean ;

%mean
```

Note that when there are no parameters there are no parentheses in either the definition of the macro or the invocation. The example is very limited because it produces a means report of only one data set and you probably don't even have a data set with that name in your program. What if we just left out the data parameter in the PROC statement? As I have already suggested, good SAS programmers don't do that. It creates a program that is too unstable and hard to read. What if we used?

```
%macro mean ;

    proc means data = &data ;
```

```

run ;

%mend mean ;

%mean

```

Well it may work, but not well. First of all there is a secret understanding that DATA must be defined outside the macro. Second of all it is not clear where to look for the value of DATA. It could be anywhere before the macro is invoked. Parameters should be thought of as part of a contract between the macro and the consuming code. If you provide good values for these parameters then this macro will generate code to do something. Consequently the parameter list should be complete and hold all requirements. With parameters it is clear what responsibilities the consuming code has and therefore the consuming code is easier to read. Without parameters there is no contract and it is hard for the reader to know whether the hidden requirements have been met or not. In small simple cases one might get away with this style of programming, but as soon as the problem gets big and complex the program will be very hard to control.

Now suppose you did write a useful macro without parameters, and have used it in lots of places. What happens when you want to add a new feature? If you add a new secret understanding than all existing code must be changed to meet the new requirement. If you decide now to make it a parameter then again all existing code must be changed to include parentheses in the macro invocation. In general every useful macro will acquire new duties requiring new parameters, hence it is never worth developing macros without parameters.

Is TPRINT finished? No, very quickly you will discover that it may be embarrassing to print a whole data set. So how should the print be limited? Again we find that the macro cannot know. On the consumer knows whether LIB.MYDATA is trivial enough that twenty observations would do, or significant enough that 500 observations or even the whole data set will be needed. You see we had better add a parameter to find out. Of course we should call the parameter OBS and give it a default value. Twenty and five hundred give two useful extremes.

Is TPRINT finished? No, maybe someone wants to specify the variables to be printed, or use by processing. You should be able to think of many more good parameters that would make the macro more useful.

Was the design good? I think the best test for a program is what I call the jiggle test. Make a minor change in what the code should do. Is the resulting change in code commensurately small? If so then the design is good. TPRINT just passed its first test.

Macro Decisions

Although we have considered only two macro instructions, the %LET and the %MACRO statements, we still have a considerable amount of power to add to plain SAS programs.

Now let's turn to making decisions. In macro code that is the %IF statement. It must be made in a macro in such a manner that the macro compiler will see it. The form is

```

%if condition %then consequent ;

```

So what is the difference between a SAS IF statement and a macro %IF statement? The %IF statement is executed during the generation of SAS code. Typically it is a decision about what code to generate. This is quite different from a SAS IF statement, which is deciding whether to execute a block of compiled code for a given data situation, not whether the block of code should be compiled.

The consequent must be is a single macro instruction, but %DO/%END can be used to indicate a block of macro instructions. Suppose we want to extend the usefulness of our TPRINT macro by giving it the ability to print or not print the data set. Let's assume a macro variable DEBUG has been assigned at the top of the program with a value Y or N. Then we can make the decision based on this value. Here is the code.

```

%macro tprint (data=&syslast,
              obs = 20 ,
              t1=3) ;

  %global debug ; /* value Y or N */

  %if %upcase(&debug) = Y %then
  %do ;
    title&t1

```

```

        "Data set: &data(obs=&obs)" ;
    proc print
        data = &data (obs=&obs);
    run ;
    title&tl ;
%end ;

%mend tprint ;

```

In the previous section I suggested that all communication between the macro and consuming code should be via parameters, but here I am breaking the rule and suggesting that DEBUG be an external macro variable. Why? First of all we do not have enough tools. Normally I would have a parameter to name the outside variable so it becomes part of the contract. Secondly, this is a very exceptional case. DEBUG is really a program parameter and it is very convenient to be able to control debugging activity at the program level. Hence I am breaking the rule. However, I added a %GLOBAL statement to the macro to warn the reader that this explicit variable is being imported from the outside. In general you should keep a very tight rein on global variables. In fact this is the only one I usually allow in my programs.

At this point TPRINT has become a really useful little macro. It automatically documents what it prints and it can be turned on or off in one place at the top of the program. Hence it is quite useful for developing large complex programs even in this rather simple form. With more %IF statements it is possible to give TPRINT every option that PROC PRINT has and more.

In the example above it is clear that an IF statement would not do, since IF statements must be in DATA steps and you cannot have procedure code inside them. But, when in a DATA step the situation is less clear. Suppose we have a macro MKTRANS that should subset a daily data set on Mondays and Thursdays. On Mondays the data set is called WORK.MONDAY and on Thursdays it is called WORK.THURSDAY. Could we use a DATA step IF?

```

if weekday(today())=2 then
    set Monday ;
else
if weekday(today())=5 then
    set Thursday ;
else
do ;
    put "Program can only be run on "
        "Mondays and Thursdays" ;
    abort ;
end ;

```

No! On Mondays the statement

```
set Thursday ;
```

will not compile and on Thursdays the statement

```
set Monday ;
```

will not compile. We cannot decide which SET statement to execute, we must decide which SET statement to generate. It must be a %IF instruction.

What about the following attempt?

```

%if weekday(today())=2 %then
    set Monday ;

```

There are two basic mistakes here. First the condition is wrong. On the left hand side we have the words

```
weekday(today())
```

How could these words ever be a single digit. Oh, no, you say, they are functions to be evaluated. When? They are DATA step functions to be evaluated when the DATA step executes. But the %IF statement's condition is evaluated during macro execution time before the code is even generated. Remember that was what we were trying to decide, which code to generate.

There is a second problem. Who owns the semi-colon at the end of the second line? Well it belongs to the macro facility for ending the %IF statement. The rule is that when the macro compiler is looking for the end of a macro instruction the first semi-

colon encountered is that end. Thus it will not be passed on to SAS and the SET statement has no semicolon ending it. We could use the %DO/%END construction to get around the second problem. For now let's add a parameter asking the consumer what day it is. This is not a good solution but we will fix it later. Here is the macro with "..." to indicate code left out.

```
%macro mktrans (day=1) ;
...
data trans ;
  %if &day = 2 %then
  %do ;
    set Monday ;
  %end ;
  %else
  %if &day = 5 %then
  %do ;
    set Thursday ;
  %end ;
  %else
  %do ;
    put "Program can only be run"
      "on Mondays and Thursdays";
    abort ;
  %end ;
  ...
..run ;
%mend mktrans ;
```

Do you notice that the code is beginning to get kind of hard to read. Why is that? The macro instructions are intertwined with the SAS code. If we made a macro variable and decided on the name outside the DATA step it would be clearer.

```
%macro mktrans (day=1) ;
  %local data ;

  %if &day = 2 %then
    %let data = Monday ;
  %else
  %if &day = 5 %then
    %let data = Thursday ;
  %else
    %put Program can only be run ;
    %put on Mondays and Thursdays ;
    %goto mexit ;
  %end ;

  ...
data trans ;
  set data ;
  ...
..run ;

%mexit:
%mend mktrans ;
```

Here the code is clearer because the first part decides what name to use and the second part show the structure of the DATA step, which got hidden when all the macro code was put inside it. Note we just used the technique introduced in the "Macro Variable" section to lift the problem out of the DATA step and place it earlier in the code.

We also sneaked in two new tools.

The %LOCAL statement declares that the variable DATA is local to the macro, in other words, it has nothing to do with any outside variable called DATA and it will cease to exist when the macro stops generating code. It is a good habit to declare

variables local. If you do not use %LOCAL the macro facility first looks outside to see if it can find the variable anywhere. If it does then it uses that variable and doesn't create a local one. If it cannot find the variable then it makes a new local variable. This can cause instability, because one execution of the macro might not find the variable and a later execution might. Of course, if you are not finished using the outside variable then changing its value can be disastrous.

I created the above problem to illustrate a point, an important technique on how to separate intertwined macro instructions and SAS code.

Why did we have to ask the user what day it is? You would think the system ought to know. In fact, it does. There is an automatic variable, SYSDAY, giving the day in words. Using of this fact, and the %INDEX function we have:

```
%macro mktrans () ;

    %if not %index(Monday Thrsday,
                  &sysday)
    %then %do ;
        %put Program can only be run ;
        %put on Mondays and Thursdays ;
        %goto mexit ;
    %end ;

    ...
data trans ;
    set &sysday ;
    ...
..run ;

%mexit:
%mend mktrans ;
```

The %INDEX function, like the DATA step function, returns the position where the second string is found in the first string. If the second string is not in the first, then %INDEX returns 0. Remember 0 is false. The difference is that %INDEX works on macro execution time when the code is being generated rather than at SAS execution time after the code has been compiled.

Now let's add some subsetting based on a parameter TYPE which should be a letter or nothing. If it is a letter then we want a subsetting IF statement otherwise there is no subsetting, i.e. the macro should do just what it does above. There are problems about how to make the decisions and how to indicate nothing.

The simplest way to indicate nothing is to find out how long it is using the %LENGTH function. If the length is 0 then the variable has the null value (i.e. nothing).

This problem involves two decisions, one of each kind. Do we need a subsetting IF statement and that subsetting IF statement. The first decision must be a %IF because we have to decide whether code should be generated or not. Then the code generated happens to be a form of DATA step IF because we want it to act during SAS execution time, deciding what records to subset to.

```
%macro mktrans (type=A) ;

    %if not %index(Monday Thrsday,
                  &sysday)
    %then %do ;
        %put Program can only be run ;
        %put on Mondays and Thursdays ;
        %goto mexit ;
    %end ;

    ...
data trans ;
    set &sysday ;
    %if %length(&type) > 0 %then
    %do ;
```

```

        if upcase(type) =
            "%upcase(&type)" ;
    %end ;
    ...
..run ;

%mexit:
%mend mktrans ;

```

Now let's look closely at the new code

```

        if upcase(type) =
            "%upcase(&type)" ;

```

On the left hand side we have "upcase(type)". Here TYPE is a variable on the input data set so there is no ampersand and UPCASE is the DATA step function acting on that variable to guarantee a standard of comparison. On the right hand we have the macro variable. It also needs to be upper cased. When? Once, when the code is generated because it is a parameter and will not be changing during the DATA step execution. Hence we use the %UPCASE macro function. Finally double quotes are needed. Why? The DATA step requires them; otherwise the compiler would think the letter generated is a variable name. Remember we cannot use single quotes because then the macro facility would not receive the macro reference to resolve it.

Macro Looping

Now what if you want to repeat code? There are two forms of looping - the iterative do-loop and the conditional do-loop. the iterative form is

```

%do variable = firstvalue %to
    lastvalue %by increment ;
...
%end ;

```

This code must be macro compiled, i.e. it must be inside a macro and in a form seen by the macro compiler. The *firstvalue*, *lastvalue*, and *increment* can be either literal values or macro expressions to be evaluated. The expressions are evaluate once at the beginning of loop execution. Hence changes to the variables in these expressions inside the loop will have no affect on execution of the loop.

Suppose we want a macro to generate lists with a common root and number on the end. Here is the code.

```

%macro mklist(root=ds,from=1,to=1) ;
    %local i ;
    %do i = &from %to &to ;
        &root&i
    %end ;
%mend mklist ;

```

Note that there is no semicolon on the line in the loop. Why? We have in mind lists of variables or data sets and SAS would get upset if we started sticking semicolons in these lists.

Now lets use the macro to concatenate 50 data sets with names FILE1, FILE2, ... FILE50.

```

data all ;
    set %mklist(root=file,to=50) ;
run ;

```

See, it is quite simple. What about the semi-colon on the second line, who owns it? SAS of course, remember that macro invocation does not use a semi-colon. Can we call a macro inside a SAS statement? Sure, why not, as long as the code doesn't generate something illegal?

Suppose we wanted to print 20 observations from each of the sets in the problem above. Here is the code.

```

%macro mkprints ( root = ds,
                num = 1,

```

```

                obs = 20,
                t1 = 3
            ) ;
%local i ;
%do i = 1 %to &num ;
    %tprint ( data=&root&i, obs=&obs)
%end ;
%mend mkprint ;

```

Hey, where did that TPRINT come from? Is it fair to use it? Sure, remember we made it in the section on macros. The whole purpose of designing good macros is to be able to use them in other good macros.

Now it should be clear why we have been so fussy about using parameters and insuring that variables are local. Lots of things can go wrong when you don't follow these practices and have macros calling macros that are written by other programmers or even yourself last month.

Let's look at one simple example. Suppose we have a macro WRTBLOCK which simply writes 1 2 3 ... up to some number n, and a macro WRTREP which calls WRTBLOCK some specified number of times. Here are the macros (without protection!).

```

%macro wrtrep ( nblocks=5, nnum=5 ) ;
    %do i = 1 %to &nblocks ;
        %wrtblock ( n = &nnum ) ;
    %end ;
%mend wrtrep ;

%macro wrtblock ( n = 3 ) ;
    %do i = 1 %to &n ;
        %put &i ;
    %end ;
%mend wrtblock ;

```

The macro WRTBLOCK was tested and worked fine. The macro WRTREP was also tested using a stub macro, i.e. a macro that just puts out a message saying it is executing, and it also worked fine. Now we put the two working macros together and what happens.

The call

```
%wrtrep ( nblocks = 2 )
```

did not work because only one block was written.

The call

```
%wrtrep ( nblocks = 7 )
```

did not work because it would have run until the log was filled.

What went wrong? WRTBLOCK is using the variable I owned by WRTREP.

In the first case, in the first call to WRTBLOCK, I is incremented from 1 to 5 (ending at 6). Since 6 is greater than 2 the do-loop in WRTREP stops prematurely after the first iteration.

In the second case, in each call to WRTBLOCK, I is incremented from 1 to 5 (ending at 6). Since 6 is less than 7 the loop in WRTREP iterates again. Since 6 is always less than 7, WRTBLOCK is called repeatedly until the log is filled or you cancel.

Disaster has struck and the code is just a few lines long! Each part was tested and worked, but the combination is impossible. Imagine facing this mess in a real situation with hundreds of lines of codes instead of just 10 lines .

If each macro declares its variables local then everything works fine as intended. A macro should not usually need to mess with variables in a calling macro, and when it does, it should be given explicit permission to do so through the parameter interface.

Summary

We haven't covered all the information about macro and we haven't even covered all the principles of good design, but we have made a start. With what you have learned in this tutorial you should be able to begin writing good simple macro systems and with that experience you should be able to pick up any new macro tools not covered here. The foundations is sound.

Now let's take a quick look at the answer to the question in the abstract from my programming professor. Macro is too hard to develop and test to use anywhere. What kind of programs should use macros? Programs that will be run many times or require the flexibility that the macro facility provides. Often in DATA step programming, arrays provide a better answer than macro.

Summary

The author can be contacted by mail at:

149 Kendal Drive
Kennett Square, PA 19348-2045

or by e-mail at:

iw1sas@gmail.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration