Macros from Beginning to Mend
A Simple and Practical Approach to the SAS® Macro Facility

Michael G. Sadof, MGS Associates, Inc., Bethesda, MD.

ABSTRACT

The macro facility is an important feature of the SAS Language. Mastering this facility will enable the novice user to become an accomplished programmer. This tutorial session will use a building block approach taking the user from the most basic %LET statements to parameter driven, code generating macros that can save time and effort. In most SAS applications for the social sciences or business there is a great deal of data manipulation to prepare for the tasks at hand whether it involves reporting, data warehousing, or statistics. The use of Macros will reduce the repetitiveness of manipulating data and will facilitate uniform and easy maintenance of the system. This paper will use step by step examples to show the user how to use macro substitution to provide simple yet powerful SAS code. The user will learn how to create variables on the fly, and how to use the macro facility to generate code. We will explore the way macros relate to SAS code and how to avoid common errors and easily debug macros.

INTRODUCTION

This paper will discuss an easy way to get started with the SAS macro facility. It will help the novice SAS user bridge the gap between beginner and expert programmer. The SAS® Macro Language: Reference, First Edition states that: “The macro facility is a tool for extending and customizing the SAS system and for reducing the amount of text you must enter to do common tasks.” It is a method of creating shortcuts in SAS programs. It enhances the SAS programming facilities and works with existing code. Macros can be used in the data step, procedures, and Screen Control Language (SCL) code. It works with all operating systems and comes as part of base SAS. It can be used a way to create streamlined code. It will not necessarily make code in the data step run faster or more efficiently. In some cases it may slow down the program due to the need to compile code at execution time. However it will reduce the amount of coding you actually have to write to accomplish data tasks. It will provide for easier maintenance, parameter driven, modularized code. It is true that the code looks more complicated and requires more testing but once you learn it, it will be easy and you will learn to rely upon it and enjoy using the SAS macro facility. This paper will not cover every aspect of the macro facility. It will however assist the novice user begin to use macros.

%LET

There are several types of macro statements. The simplest, but very useful, form of the macro language is the assignment statement. The assignment statement in is simplest form looks like:

%LET macvar = text_string;

This statement creates a macro variable named: macvar and assigns it the value of text_string. Under most conditions (some of which will be discussed later) this newly created macro variable is available to every DATA step and PROC step to follow in the program. It must be referred to as &macvar and will evaluate to whatever is in text_string. First let us see how you use the macro variable in a program. Since we are all familiar with the title statement it will be used to demonstrate the evaluation of macro variables in programs. Please note that quotation marks are very significant in working with macros but there are a few tricks that will help you code efficiently without errors. Consider program fragment:

%LET city=Baltimore;
....
....
TITLE "This is &city";

When the title statement executes it produces the title:

This city is Baltimore

Double quotes (”) rather than single quotes(‘) are significant and necessary and the title statement with single quotes (‘) would evaluate with an undesirable result at execution time as:

This city is &city

If you use quotes when creating a macro variable in an assignment statement the quotes will be saved as part of the macro as follows:

%LET city=’Baltimore, MD’;
....
Title "The address is &city";
The address is ‘Baltimore, MD’

It generally a good idea not to use quotes when assigning macro variables with the %LET statement but rather use them as needed when recalling the macro value. Once assigned in a program (but not
necessarily within a macro definition) the %LET statement creates a GLOBAL variable that is available for your use almost anywhere in your program after the assignment statement. The macro is recalled by placing an ampersand(&) before the macro name. To insure that a macro variable will be available for the entire SAS session you may mention the variable in a %GLOBAL statement. Before we go into the scope of macro variables it is important to look at some examples.

The most common uses for macro variables are as data file names or targets of logical-if statements. Consider the code:

```
%LET filemac = BALT;
%LET city = Baltimore;
%LET select = 14;

DATA &filemac; SET scores;
IF age = &select;
IF city = "&city";
```

Produces a data set named BALT with information from the scores data base where age = 14 for the city of Baltimore. Let us assume that age is a numeric variable. Notice that quotes are not used because we want the statement to evaluate as:

`IF age = 14 ;   rather than IF age = "14";`

to prevent numeric to character conversion. However double quotes are used in the IF “&city” statement because we want it to evaluate as:

```
IF city = "Baltimore";
```

A simple trick which works most of the time concerning the use of quotes is as follows:

1. Do not use quotes in assigning macro variables with the %LET statement.
2. When using the ampersand variables in code use quotes if the expression should evaluate to a character.
3. Do not use quotes if it should evaluate to a number.
4. Do not use quotes when invoking macro variables as file names.

In its most basic form the %LET assignment statements assigns a series of characters to a macro variable and places that GLOBAL variable into the global symbol table and makes that GLOBAL variable (or colloquially referred to as amper-variable) available to the rest of the program.

What is the advantage of using the %LET statement if it only substituting character strings in other places in the program? The macro variable will allow you to structure your programs in a more readable and maintainable fashion. Consider multiple title statements of with various contents scattered around in many places in the code along with multiple 'select ifs' or 'where clauses'. By placing the %LET statements at the beginning of the code you can easily change cities or selections throughout the program. As we build upon the simple macros you will see how important they are.

**CALL SYMPUT**

Before we go on to macro definitions let us remember that the %LET statement assigns essential static (non data step dependent) text to macro variables. However to put data from a DATA STEP into a macro variable you will use a macro function "SYMPUT". It looks and sounds complicated but it is really very simple. This function takes a data value from a data step and puts it into the GLOBAL or LOCAL macro buffer. The syntax to get the state from a DATA STEP into an amper-variable named state is as follows:

```
DATA one; SET two;
WHERE zipcode='12345';
CALL SYMPUT ('mystate',state);
RUN;

PROC PRINT;
TITLE
 "For zipcode = 12345 the state is &mystate";
```

The RUN statement is required in this case. This code will extract the state value for a particular zipcode (assuming zipcode and state are variables in the SAS data set two) and place it in a GLOBAL variable for your later use. The evaluation will take place each time a record is read but only the last value will be maintained. So if there are several records in data set one only the last occurrence of state in zipcode ‘12345’ will be kept. The syntax of CALL SYMPUT looks slightly strange at first. Remember to use single quotes surrounding the macro variable name as ‘MYSTATE’ and place a variable name or expression as the second parameter. In the previous example the variable STATE was used. As you become more familiar with the macros and the CALL SYMPUT function you will use expressions and formats in place of simple variable names. The following expression will create a macro variable &mydate that can be used in title statements or reports. The result is a formatted date that can enhance the look of output. You must use the “&sysdate’d function since the automatic macro variable &sysdate is in the form 08OCT97.
The following code will convert &sysdate to a longer more readable format:

```sas
DATA _NULL_
   CALL SYMPUT('mydate', TRIM(PUT("&sysdate"d,worddate22.)))
RUN;
%PUT &mydate;
```

Result: October 8, 1997

%PUT

Another statement that is quite simple but very useful is the %PUT statement. It may be used virtually anywhere and will write to the log any values of user defined or system defined macro variables such as:

%PUT
"The City is &city and the state is &state on &sysday";

This %PUT statement will write to the SAS log:

"The city is Baltimore and the state is Maryland on Tuesday"

Notice the use of an automatic system generated macro variable &sysday. They can always be used in code by preceding the name with an ampersand as in: &sysdate to get the current date. Several keywords may be used with the %PUT statement to write all or part of the GLOBAL symbol table. These are very helpful in debugging and testing macros within programs.

To write all macro variables to the log:

%PUT _ALL_;

To write all user defined global variables to the log:

%PUT _GLOBAL_;

To write all user defined local variables (those built inside macro definitions).

%PUT _LOCAL_;

To write all system defined _automatic_ variables to the log:

%PUT _AUTOMATIC_

To see the effect of these %PUT statements just place them anywhere in your code and submit. The evaluation of the macro will be written to the SAS log next to the name of the variable.

MACRO DEFINITIONS

Now that you are familiar with GLOBAL variables such as &city, &sysday (a system defined variable), and &filemac let us go on to developing macro definitions which can be used as subroutines in your program that can be executed at various places and under various conditions. The most simple form of the macro definition begins with %MACRO and ends with %MEND.

```
%MACRO firstmac;
   %PUT Hello World - this is my first 'Macro';
%MEND firstmac;
```

This macro can be executed or called in a program with the following line:

```
%firstmac;
```

Remember a macro definition begins with %MACRO `mac_name`; and ends with %MEND `mac_name`; The invocation of that macro somewhere else in the code is %`mac_name`. The macro must be defined before it can be invoked or executed in a program.

Note here that macro definition executions use the percent(%) in front of the name and macro variables use ampersands (&). For uncomplicated macros it is generally easier to use the amper-variable rather than the %MACRO definition however as you build more intricate macros the functions of the definition style macro become more evident. Within the macro definition (between the %MACRO and the %MEND) you can place any valid SAS code statements or statement fragments. The macro name (%`mac_name`) will expand out when it is executed. Consider the following sample which will create code shorthand for your programs:

```
%LET avars  =  height weight age city ;
%MACRO bvars ;
score1 score2 midterm final
%MEND bvars ;
```

```
DATA one; SET two (KEEP= &avars %bvars );
PROC SUMMARY;
   CLASS &avars ;
   VAR %bvars ;
   OUTPUT OUT=three SUM=;
PROC PRINT DATA=three;
   VAR &avars %bvars  ;
RUN;
```

In the above example it is shown that the %LET statement and the %MACRO statement are similar.
Both are used to define a string of variables that will be used later in the program. Notice semicolons (;) are not used inside these macro definitions because we are trying to generate strings of variable names not full statements. The main difference between the two forms of macros (amper-or percent-) is the way they are invoked. When invoking the macro defined with the %LET statement you must use an ampersand (as in &avars) while the %MACRO-%MEND definition uses a percent sign (as in %bvars). You can place either version in various places inside the program to save time and assist in maintaining or modifying programs. Either form is essentially equivalent and really suggest a type of programming style rather than a correct or incorrect method. It is generally advisable to use the %LET statement for simpler definitions and reserve the %MACRO-%MEND construct for more involved definitions. A good rule of thumb is to use the %LET statement for single variables and the %MACRO-%MEND definition for complicated or just long definitions. An easy way to build macros and remember the %MEND statement is to start the macro by coding:

```plaintext
%MACRO mymac
%MEND mymac;
```

and then fill in the center. In this manner you can never forget the %MEND. I always do the same for if-thens, and do-ends. To say that I never get the 'Unclosed DO blocks' error message would be lying but it does cut down on errors.

Parameter Driven Macro Definitions:

The next step is to introduce parameters into the macro definition which will make them more flexible and pave the way for data driven selection of code. Parameters are values that are passed to the macro at the time of invocation. They are defined in a set of parentheses following the macro name. When referred to inside the definition they need preceding ampersands. It is easier to visualize in code:

```plaintext
%MACRO SECOND (XX,YY,ZZ);
  %PUT Call second mac with &xx &yy &zz;
  PROC PRINT;
  VAR &xx &yy;
  SUM &xx &yy;
  FORMAT &xx &zz. &yy 5.0;
  %MEND SECOND;
```

To invoke the macro use the following:

```plaintext
%second (age,height,3.0);
```

Notice you can place variable names or values in the parameter list when calling the macro. When executed the &xx, &yy, and &zz are replaced with the variables and literals you called the macro with. When the program executes the processor sees the code:

```plaintext
%PUT calling macro second with age height 3.0;
proc print;
var age height;
sum age height;
format age 3.0 height 5.0;
```

Note that the format statement evaluates correctly with the inclusion of a period (.) after the &zz in the code.
You can invoke the macro again with:

```plaintext
%second (weight,age,4.2);
```

to produce another section of code with different parameters. Remember that the macro definitions produce 'code'. They do not execute until they are invoked in the appropriate place in the program with the %MACRO-name invocation.

SCOPE

When the %LET statements are placed in open code (outside of any DATA step or %MACRO definition) the variables they define become GLOBAL. That is to say they are available to any and all statements that follow. When a %LET statement is found within a %MACRO definition and the same variable has not been defined before the variable becomes LOCAL to that %MACRO definition and is not available outside of that macro unless it is defined with the %GLOBAL statement such as:

```plaintext
%GLOBAL x;
```

The exact hierarchy of global and local variables and how the buffers are maintained is complicated but be aware that if you defined a variable and then get the message: “Apparent symbolic reference not resolved” then the macro might have been defined locally and is not available in the GLOBAL symbol table at the time of invocation. Be sure the macro was spelled correctly. Be sure there are run statements after any CALL SYMPUT’s, and be sure to mention the macro in a %GLOBAL statement if it was defined inside a %MACRO definition.

Expanding Macro Definitions

In the previous example you can see how parameters can be used to execute code differently in different places in your program. Now let us see how we can build macros that will create differing code depending upon the data. For instance let's say you want to execute a PROC PRINT if there is data in the data set
but print an error message if there is an empty data
set. Or you might want to execute a frequency for one
type of data or a means for another type of data. You
can build if-then logic into the macro definition thereby
selectively building code. Notice that in the following
example several statements are outside the if-then
logic an will execute for any value of &type; The
correct syntax for if-then logic within a macro requires
a percent (%) before the following elements: (%IF,
%THEN, %ELSE, %DO, %END). This tells the macro
processor to interpret the %IF's as macro statements
and conditionally generate program code.

%MACRO thirdmac ( type , title );
%PUT Begin third mac with type: type and title: &title ;
DATA three; SET one ;
IF dat_type = &type;
%IF &type = 1 %THEN %DO;
PROC PRINT;
TITLE "report for: &title";
%END;
%ELSE %DO;
PROC MEANS;
TITLE "Sample Statistics for: &title ";
%END;
RUN;
%PUT Macro thirdmac now finished ;
%MEND thirdmac;

To execute or invoke this macro you might use the
following statement to invoke a PROC MEANS on the
data set three:

%thirdmac(2,This will run a proc means);

PORTABILITY

So far we have discussed simple code and simple
invocations of macros which may well have been
performed without macros and may be just easier to
replicate code instead of trying to be fancy. Now let us
begin to look at situations where macros really make
things easier. On a recent project I was developing
code that had to run both on several Unix machines
(two flavors of Unix) and under windows 3.1. No
problem for the SAS System, you say, just remember
to change the file names. Well when you have
hundreds of files it becomes a problem. What we did,
however, was to develop a series of macros which
determined which system we were running on and
build the file names accordingly. The macros were
tricky and we had to do a lot of testing to get them to
work properly in every situation but it saved many
hours of changing libnames and filenames each time
we had to run since the programs became truly
portable. Here is a simplified example:

%MACRO nameit ;
%LET path = mypath;
%LET file = filename;
%IF &sysscp = WIN
  %THEN %LET DLM = \;
  %ELSE %LET DLM = / ;
%END;
%IF &sysscp = WIN
  %THEN %LET DIR = c:\sas ;
  %ELSE %LET DIR = m:/myown/unix/direc ;
%LET LIBNM = &dir.&dlm;
%LET FLNM  = &dir.&dlm.&path.&dlm.&file.. dat;
FILENAME file1 "&flnm";
LIBNAME  lib1  "&libnm";
%MEND nameit ;

Notice the period(.) after the invocation of the &dir
variable. This is necessary to indicate the end of the
local variable &dir and the start of the next variable
&dlm. If you actually want a period in the file name
such as filename.ext you must use two periods as in
the FLNM definition. When invoked the macro
%nameit will produce the correct libname and
filename statements depending upon the operating
system. The double quotes will enable the macro to
evaluate properly in a SAS program which requires
filenames to be in quoted strings.

For Windows:
FILENAME file1 "c:\sas\mypath\mydata.dat ;
LIBNAME  lib1 "c:\sas" ;

For Unix:
FILENAME file1
  "m:/myown/unix/direc/mypath/mydata.dat";
LIBNAME   lib1 "m:/myown/unix/direc/";

MACRO ARRAYS

As we have seen the %DO and %IF statements in the
macro facility enable us to create long statements and
code with little effort. The %DO statement has various
forms (%DO-%WHILE, %DO-%UNTIL) and the
iterative %DO as shown below:

%MACRO arrayme;
%DO i = 1 %TO 5;
  file&i
%END;
%MEND arrayme;

data one; set %arrayme;
The macro evaluates to the following at execution time:

DATA one; SET file1 file2 file3 file4 file5;

In this macro we generate a list of 5 file names. The values of 1 to 5 are substituted in the expression file&i to produce (file1 file2 file3 file4 file5). The above code will set five files into data set one.

DOUBLE AMPERSANDS:

In our previous example our files were named file1, file2, file3, file4, and file5. Let us suppose that our files are named for the state data that they contain. It will be necessary to build code that has amper variables in it. If we have 5 macro variables defined as follows:

```sas
%LET a1=KS;
%LET a2=MD;
%LET a3=CA;
%LET a4=NY;
%LET a5=KY;
```

We can generate code that will set all these files into one file as before but with the correct file names.

```sas
%MACRO stname;
%DO i = 1 % to 5 ;
&&a&i
%END;
%MEND stname;
```

DATA one; SET %stname;
The first pass of the macro processor creates the variables &a1-&a5 and the second pass evaluates the amper-variables to their text strings (KS MD CA NY KY). We actually want to build code that looks like:

```sas
&a1 &a2 &a3 &a4 &a5
```

The macro processor evaluates each occurrence of multiple ampersands on a separate pass so the double ampersand give the desired effect.

Consider this example of census data and its solution. We have a data file of several million records containing the names, information and zipcodes of residents a region of the US. We want to merge this with zipcode information. The zipcode information is divided into 50 files, one for each state. You guessed it the file names are the names of the states like AZ, KS, NY etc. These zipcode files are too big to set together and merge with the regional resident file so we use macros to select only those state zipcode files of the states that are contained in the regional residents file. Of course there are many approaches but this is just one using macros.

```sas
data _null_; set region(keep=state) end=eof;
length st_mac $ 150 ;
retain st_mac "" num_st 0 ;
if index(st_mac,state)=0 then do;
   num_st=num_st+1;
   st_mac=trim(st_mac) || "" || state ;
end;
if eof then call symput ('st_mac',st_mac);
run; * must put run after data step with call symput ;
```

```
** now we can select only state zipcode files we need *
```

data regzip; set &st_mac  ;
proc sort; by zip;
data region; merge region regzip; by zip;
```

The DATA _NULL_ step builds a string of state file names with each new state it reads from the region data set. It checks with the index function to be sure it has not already added that state to the string. The amper-variable &st_mac becomes a string of unique state names like (AK KS CA NY) depending upon which states are in the regional residents file. You must set the initial length to 150 to leave enough room for 50 states. Also you must trim the result before added a new state otherwise the string will be greater than 150. In this manner we can build macro strings based upon the data without knowing ahead of time what states are in the data set. As we combine these various techniques we can build programs that generate code based upon the incoming data.

CONCLUSION

We have seen how you can use the SAS Macro Facility to enhance your programs. Starting with the %LET statement and advancing to macro definitions we have learned that macros enable you to build complex and repetitive code with ease. Macro variables and macro definitions must be defined before they can be invoked. Macros are essential strings of data saved in a buffer available to every data step after they are defined. The execution of macros in code is simply a replacement of the code generated by the macro definition at the point where it is invoked. The use of macros in SAS code can streamline your programs and enable you to modularize your code. It will assist you in developing parameter driven and data driven programs that will require less maintenance. I hope this short presentation will assist you in developing elaborate and convenient reusable
code. The use of macros should only be limited by your imagination.

REFERENCES


Michael G. Sadof
MGS Associates, Inc.
6203 Yorkshire Terrace
Bethesda, MD 20814
(301) 530-2353

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

Other brand and product names are registered trademarks or trademarks of their respective companies.